



Faculty of Science and Technology

BSc (Hons) Games Software Engineering

May 2021

LIBRARY FOR IMPLEMENTING REAL TIME RAYTRACING

by

Cosmin Apopei

DISSERTATION DECLARATION

This Dissertation/Project Report is submitted in partial fulfillment of the requirements for an honours degree at Bournemouth University. I declare that this Project Report is my own work and that it does not contravene any academic offence as specified in the University's regulations.

Retention

I agree that, should the University wish to retain it for reference purposes, a copy of my Project Report may be held by Bournemouth University normally for a period of 3 academic years. I understand that my Project Report may be destroyed once the retention period has expired. I am also aware that the University does not guarantee to retain this Project Report for any length of time (if at all) and that I have been advised to retain a copy for my future reference.

Confidentiality

I confirm that this Project Report does not contain information of a commercial or confidential nature or include personal information other than that which would normally be in the public domain unless the relevant permissions have been obtained. In particular, any information which identifies a particular individual's religious or political beliefs, information relating to their health, ethnicity, criminal history or personal life has been anonymized unless permission for its publication has been granted from the person to whom it relates.

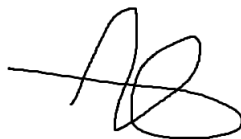
Copyright

The copyright for this dissertation remains with me. dissertation remains with me.

Requests for Information

I agree that this Project Report may be made available as the result of a request for information under the Freedom of Information Act.

Signed:

A handwritten signature in black ink, consisting of a series of loops and a long horizontal stroke extending to the left.

Name: Cosmin Apopei

Date: 14/05/2021

Programme: BSc GSE

1 Acknowledgements

I would like to thank my supervisors Leigh McLoughlin and Karsten Pedersen for the continuous support and guidance provided during the creation of this project. I believe that my weekly meetings with Leigh have been of incredible help in staying motivated and maintaining a decent workflow during this unusual pandemic times.

I would also like to thank my family for supporting me during the development of this process. Their encouragements and help with relieving me of day to day duties have allowed me to bring this project to a much higher standard than time would've otherwise allowed me.

2 Abstract

Ray tracing as a rendering technique has always been preferred for achieving high degrees of realism in computer graphics. However, its high computational cost have made it impossible to use Ray tracing in real time for a long time, which made it not viable for video game graphics.

Recent technological advancements have allowed consumer grade equipment to finally be able to support real time Ray tracing; this lead to major game companies trying to integrate Ray tracing bound features, such as real time shadows or real time reflections, into their products. This process is still very expensive and difficult, which, when coupled with the limited amount of consumers that could appreciate this additional realism, is deterring smaller game studios from attempting it.

This project aims to create a library that allows developers to quickly and easily create an environment that enables them to start developing real time Ray tracing bound features. The library will be implemented using C++ as a programming language, and it will make use of the DXR extension for the DirectX 12 graphics API to access the features in the new graphics card architectures that make real time ray tracing possible.

Contents

1	Acknowledgements	2
2	Abstract	3
3	Introduction	9
3.1	Aims	9
3.2	Objectives	10
4	Literature review	11
4.1	Ray tracing overview	11
4.2	Ray tracing in depth	12
4.3	Ray tracing compared to rasterization	13
4.3.1	Offline renderers	13
4.4	Real time Ray tracing history	15
4.5	Ray tracing in video games	16
4.5.1	Uses of Ray Tracing	16
4.5.2	Nvidia RTX: Real time Ray tracing	17
4.6	Ray tracing APIs	21
4.7	Library design principles	21
4.7.1	S.O.L.I.D Principles	21
4.7.2	Documentation - Doxygen	22
5	Design and Implementation	23
5.1	Defining the Library	23
5.2	Compatibility with engines	23
5.3	DXR - DirectX Ray tracing	25
5.4	Library architecture	26
5.5	Initialising DirectX 12 and creating the back end	27
5.6	Creating the Ray tracing pipeline	29
5.7	Creating the acceleration structures	31
5.8	Enabling path tracing	32
5.9	Using the library	33
6	Testing	34
6.1	Implementation in the engine	34
6.2	Performance costs	36
7	Evaluation	38
7.1	Aims and objectives met	38
7.2	Future development	38
8	Conclusion	39
9	Bibliography	40

10 Appendix	45
10.1 Ray tracing features in games.	45
10.2 GPU and CPU usage report.	46
10.3 Class diagrams	50
10.3.1 RTX_Manager	50
10.3.2 RTX_Initializer	51
10.3.3 RTX_SBTGenerator	52
10.3.4 RTX_Pipeline	53
10.3.5 RTX_BVHManager	54
10.3.6 RTX_TLAS	55
10.3.7 RTX_BLAS	56
10.3.8 RTX_Exception	57
10.3.9 RTX_PathTracer	57

List of Figures

1	Example of the same scene being rendered with rasterization and Ray tracing. The Ray traced render looks more realistic thanks to shadows and reflections (Park and Baek 2021).	9
2	Screen shot of Turner Whitted's Ray tracer.	11
3	Ray casting used to render an image containing a scene object and its shadow. . . .	11
4	The Ray tracing process explained (Suffern 2016).	12
5	Generating shadows using Ray tracing. To determine whether point P is shadows, to light ray La and Lb are sent towards the light sources A and B. Light ray Lb gets interrupted so point P is in shadow from light source B (Glassner 1989).	12
6	The differences between Rasterization and Ray tracing (Nvidia 2021).	13
7	Rendering a 4K resolution image of a dragon model with reflections and shadows enabled using Blender's Eevee RT engine took more than two minutes (Iatan 2019).	13
8	Ray tracing bound features across 11 triple A titles.	16
9	Engine models across 11 triple A titles.	16
10	Turing TU102, TU104, TU106 Architecture (Nvidia 2018).	17
11	An example of navigating a BVH for a 3D rabbit model.	18
12	An example Reflection Pipeline (Barre-Brisebois 2019).	19
13	How to generate real time shadows (Barre-Brisebois 2019).	20
14	How to generate real time shadows (Barre-Brisebois 2019).	20
15	Example of part of the HTML documentation for a game engine.	22
16	RTX APIs used across eleven triple A titles.	23
17	Top level class structure of the library.	26
18	Diagram showcasing how the user interacts with the system for the initialization process. Notice that the user only ever accesses the RTX_Manager class.	28
19	Simplified view of the new DirectX Ray tracing pipeline (Akenine-Möller and Haines 2019).	29
20	Diagram showcasing the structure of a typical SBT (Lefrançois 2021). Note how the data is contiguous to the shader it relates to.	30
21	Diagram summarizing the creation of the DXR Ray tracing pipeline.	30
22	Diagram showcasing the relationship between the TLAS and the bottom-level acceleration structures it contains (Lefrançois 2021).	31
23	The different operations a command list must do each frame.	32
24	Calls needed for the library to produce a Ray traced output.	33
25	The process of initializing the library.	34
26	An output produced by using the library. It shows 4 triangles and a plane being seen through a perspective camera.	35
27	An output produced by using the library. It shows 3 triangles rotating and casting shadows on a plane. Each model is also coloured differently.	35
28	CPU usage for generating the output shown in Figure 26	36
29	GPU usage for generating the output shown in Figure 26.	36
30	Frame time and Frames per second during the generation of the output shown in Figure 27.	37
31	Overview of the RTX_Manager class.	50
32	Overview of the RTX_Initializer class.	51
33	Overview of the RTX_SBTGenerator class.	52
34	Overview of the RTX_Pipeline class.	53

35	Overview of the RTX_BVHManager class.	54
36	Overview of the RTX_TLAS class.	55
37	Overview of the Instance structure.	55
38	Overview of the RTX_BLAS class.	56
39	Overview of the RTX_Exception class.	57
40	Overview of the RTX_PathTracer class.	57

List of Tables

1	Specifications of the system used to create the output shown in Figure 32.	36
2	Project's objectives evaluation.	38
3	List of connections needed to display a textured 3D model	45

3 Introduction

Real time Ray tracing has long been considered the "holy grail" in computer graphics (Nvidia 2018), as the visual effects it can reproduce are of unmatched realism. However, due to its high computational costs, it was hard to achieve on consumer grade hardware until recent technological advancements. The lack of a standardized API and the hardware not having sufficient computational power made the process very difficult and unrealistic to yield fast enough results. Therefore, most developer resorted to rasterization as a render technique, which is significantly less resource expensive at the cost of visual fidelity (see Figure 1).

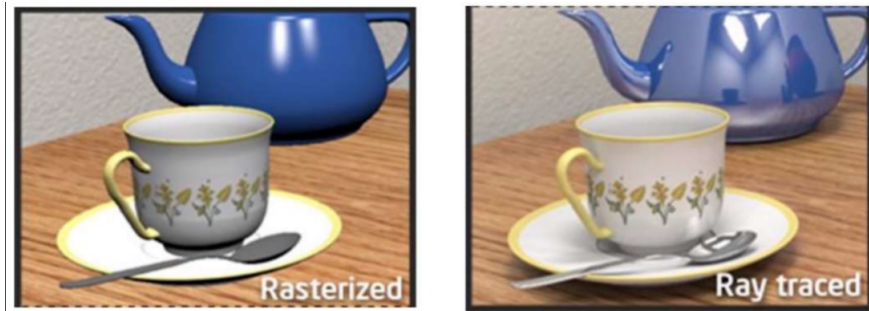


Figure 1: Example of the same scene being rendered with rasterization and Ray tracing. The Ray traced render looks more realistic thanks to shadows and reflections (Park and Baek 2021).

Ray tracing enables a number of visual effects that would otherwise be really hard to achieve in real time, such as shadows, reflections, global illumination and more. These effects are what make its visual outputs appear more realistic.

Recent technological advancements in graphic cards architecture (NVIDIA, 2018) have allowed real time Ray tracing to become a reality in video games; implementing this technology remains difficult however, since it requires a substantial amount of code to be written using a new API that many developers are not familiar with.

Game Engines based entirely on real time Ray tracing are still rare in the industry (see Appendix 1). Developers often choose to only implement specific Ray racing bound features, which often tend to be optional for the end user to enable. These high development cost for a relatively small potential value added have bound these features to only triple A studios.

This project focuses on creating a library that abstracts some of the complexities of current Ray tracing graphics APIs, with the hopes of allowing this fantastic technology to become more widely available. The project will be developed using C++, making use of the DirectX 12 graphics API and its DirectX Ray tracing extension DXR.

3.1 Aims

The project has the following aims:

- to create a library that allows developers to easily and rapidly create an environment for developing real time Ray tracing features;
- to test this library by implementing it into a game engine and producing a visual output using it.

3.2 Objectives

The following are the objectives that will ensure the aims are met:

1. conduct research into the background knowledge necessary for developing this project. This includes:
 - (a) conducting research into how Ray tracing works and how it compares to rasterization. Understanding the theory behind the rendering techniques and its benefits will help with designing a more useful library;
 - (b) examining what different Ray tracing techniques various triple A developers have implemented and what technology they have used. By understanding current trends the design of the library is more likely to be compatible with many game engines;
 - (c) researching principles and guidelines for designing effective libraries.
2. creating the library that allows developers to start working with real time Ray tracing. The steps needed for this include:
 - (a) researching and understanding the graphics APIs that will be used;
 - (b) designing a library system that respects the principles and guidelines explored;
 - (c) implementing the design using C++.
3. implementing the library into an existing system to create a visual output;
4. evaluating the implementation of the library. The evaluation will consist of:
 - (a) examining the user experience of using the library;
 - (b) ensuring that a visual output can be produced;
 - (c) checking the performance costs of the library.

4 Literature review

4.1 Ray tracing overview

Ray tracing is a rendering technique based on simulating the behaviour of light (Nvidia 2018). It produces visual outputs by tracing the path of light rays in a 3D virtual environment. The rays of light gather information regarding the objects they intersect with; this is used to calculate the colour of the pixels. This process enables the recreation of real life lighting phenomena such as reflection, refraction and shadows.

Ray tracing techniques handle the tracing of light rays by working in reverse from the camera (Caulfield 2018). This technique was first described by Arthur Appel (1968), and later improved by Turner Whitted (1980) (see Figure 2).

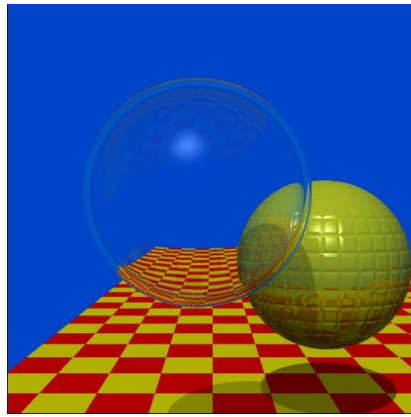


Figure 2: Screen shot of Turner Whitted's Ray tracer.

Akenine-Möller. and Haines (2019) refer to this process as *ray casting* (see Figure 3) in their book "Ray tracing Gems".

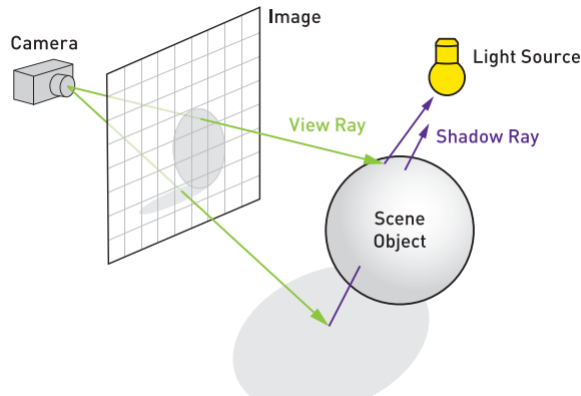


Figure 3: Ray casting used to render an image containing a scene object and its shadow.

4.2 Ray tracing in depth

The concept for backwards Ray tracing is quite simple: a Ray of light is shot from the camera towards a point in the scene. When it interacts with an object, it returns the properties of said object to create an image. These properties include roughness, albedo and reflectivity.

Almost all objects in the real world however reflect the light to a certain degree, so upon collision with an object another ray is sent towards where the light would be reflected (see Figure 4).

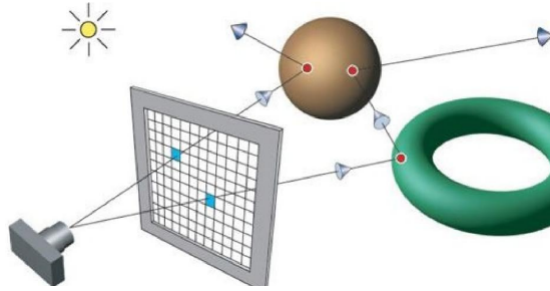


Figure 4: The Ray tracing process explained (Suffern 2016).

The rays that bounce off an object are likely to hit another object, which will then spawn another ray. This is known as recursive Ray tracing, and it is often regulated by a recursive depth parameter that determines how many times this process can occur.

To create shadows, the process is similar but instead of starting from the camera it starts from the surface that might be in shadow. A ray is sent to each light source in the scene, and if it is interrupted by any object before reaching the light source then it is in shadow (see Figure 5). This process can be refined to achieve more realistic shadows, for example by sending multiple rays in a cone towards the light source.

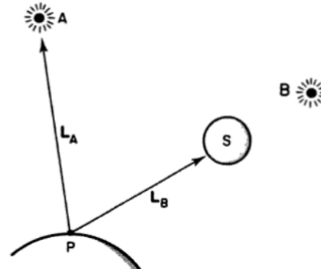


Figure 5: Generating shadows using Ray tracing. To determine whether point P is shadows, to light ray L_A and L_B are sent towards the light sources A and B. Light ray L_B gets interrupted so point P is in shadow from light source B (Glassner 1989).

When trying to generate an output for a scene which is as realistic as possible, a large amount of Rays need to be generated and traced. The most expensive part of tracing a ray is checking for intersections; to accelerate this, the data is usually stored in acceleration data structures.

4.3 Ray tracing compared to rasterization

Rasterization and Ray tracing are the two most common rendering techniques for computer graphics. Rasterization is often considered the faster approach, while Ray tracing is known to create more realistic visuals.

The two techniques work almost in an opposite way: for rasterization, the question each loop is which pixel is closest to each object, while for Ray tracing is which object is the closest to each pixel (Nvidia 2021). This difference is mainly what makes rasterization significantly faster than Ray tracing: rasterization has to perform significantly less checks than Ray tracing, and said checks are also less computationally expensive.

The figure below shows the main differences between Rasterization and Ray tracing:

Rasterization and Ray Tracing


Key Concept	Rasterization	Ray Tracing
Fundamental question	What pixels does geometry cover?	What is visible along this ray?
Key operation	Test if pixel is inside triangle	Ray-triangle intersection
How streaming works	Stream triangles (each tests pixels)	Stream rays (each tests intersections)
Inefficiencies	Shade many tris per pixel (overdraw)	Test many intersections per ray
Acceleration structure	(Hierarchical) Z-buffering	Bounding volume hierarchies
Drawbacks	Incoherent queries difficult to make	Traverses memory incoherently 

Figure 6: The differences between Rasterization and Ray tracing (Nvidia 2021).

4.3.1 Offline renderers

Ray tracing as a rendering technique has existed since commercial computers first started becoming popular; nevertheless, Rasterisation has been the staple for real time computer graphics instead (Nvidia 2018). This is because the amount of computational power and time required by Ray tracing (e.g. see Figure 7) prevented it from being able to generate real time results on most hardware.

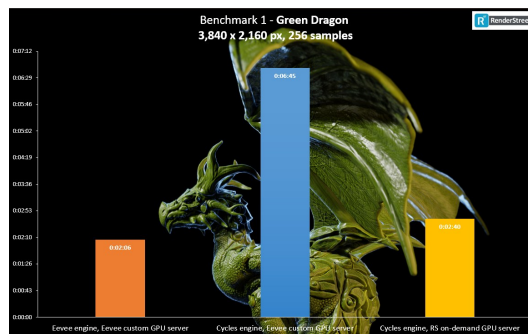


Figure 7: Rendering a 4K resolution image of a dragon model with reflections and shadows enabled using Blender's Eevee RT engine took more than two minutes (Iatan 2019).

However, Ray tracing has been used for offline rendering for a long time. This is because the high computational cost of Ray tracing does not matter as much when there is virtually no time constraint for generating an output. Offline rendering has often paved the way for discovering new Ray tracing techniques that eventually became available in real time rendering.

There are many offline renderers available to the public, some of the most popular ones being (Yamazaki 2021):

- Maxwell render, one of the most well known renderers when it comes to realism. Available in 3DS Max, Standalone, Maya and others;
- Autodesk Arnold Render, based on an advanced Monte Carlo Ray tracing integration. Available in 3DS Max, Maya, Cinema 4D and others;
- V-Ray Render, a render renown for being able to create accurate results regardless of the data load;
- Redshift, an offline renderer released in 2019 that use GPU acceleration to deliver quick results.

Arnold Render is the most popular and recognized offline renderer, especially in the movies industry where it received a Scientific and Engineering Award (Academy plaque) by the Academy of Motion Picture Arts and Sciences (SolidAngle 2017). Its deep feature list and dependable performance make it an excellent choice for many, but its complexity may appear intimidating to some beginners (Lee 2019).

Maxwell Render is most commonly used when highly realistic visuals are more important than performance. The amount of unique features that it offers, such as *FIRE* (lighting changes visible in real time) and coating (special brightness or bubbles) on the GPU (Maxwell 2021), enable for a fast working environment that its sure to produce good looking results.

Redshift is one of the newer renderers available at the moment, but its GPU acceleration for Ray tracing allows for significantly faster results. This has allowed it to rapidly grow in popularity, with some triple A game companies such as Blizzard already adopting it (Redshift 2021).

V-Ray is one of the oldest render engines available on the market, being first released in 1997 (Denham 2021). It renders on both the CPU and GPU at the same time, which makes it faster than purely CPU based engines. Its proven reliability coupled with its extensive feature list have encouraged companies to use it for massive projects, such as "Avengers: Infinity War" (RenderNow 2021).

4.4 Real time Ray tracing history

Real time Ray tracing has only become popular in video games recently, but studies into this process have existed for decades. The recent technological advancements are mainly focused on providing hardware acceleration for existing Ray tracing techniques.

An excellent example of this comes from the acceleration structure modern real time Ray tracing programs use - Bounding Volume Hierarchy (BVH). The structure was already being studied in 2005 by Christer Ericson, which described what it is and how it can be used to detect intersections (Ericson 2005). In 2013, Hapala et al. looked at how BVH traversal can be used to speed up Ray tracing (Hapala et al. 2013). In 2018, Nvidia releases RTX cards which provide hardware acceleration for BVH (Nvidia 2018).

Attempts of creating real time Ray tracers can be found from before GPU hardware acceleration was available. For example, Igno Wald has developed an interactive Ray tracer in 2001 (Wald et al. 2001) and a Ray tracer to study its use for Interactive Global Illumination (Wald et al. 2003). Igno Wald and his team also created a CPU Ray tracer for large particle data in 2015 (Wald et al. 2015). In 2006 Carsten Benthin created a real time Ray tracer on contemporary CPU architectures. He concludes that although his project offers effective parallelization and other CPU specific optimization techniques, the amount of computational power needed makes it impossible to create an output at interactive frame rates (Benthin 2006). In 2007 Johannes Gunther et al. have developed a Ray tracer on the GPU using BVH for ray traversal (Gunther et al. 2007). This project is very similar to modern Ray tracers, with the main differences being that the BVH was constructed on the CPU and transferred to the GPU, and the GPU did not accelerate neither BVH traversal nor Ray - object intersection testing.

A big step towards hardware acceleration for Ray tracing came from the introduction of Compute Unified Device Architecture (CUDA), a graphics API created by Nvidia for their graphics card in 2007 (NVIDIA 2007). This allowed developers unprecedented control over the GPU, which inspired developers to try to develop GPU sided Ray tracers.

In 2010, Martin Zlatuška and Vlastimil Havran conducted a study which examined three different Ray tracers on a GPU with CUDA. They proved that using BVH as an acceleration structure provided the best results, with it being able to render a simple 1024x1024 image in 40ms (Zlatuška and Vlastimil 2010).

In 2017 Sayed Ahmadsreza Razian and Hossein Mahvash Mohammadi have optimized a Ray tracing algorithm using CUDA, which netted them 11 frames per second using 720p resolution on a GT 840m graphics card (Razian and MahvashMohammadi 2017).

After 2018 and the release of RTX graphics cards, real time Ray tracing started being implemented by big game studios in their games (see Appendix 1). A well optimized real time Ray tracer can produce a 4K output of a simple room scene in 3.4ms, and a 4K output of large interior scene in 13.4ms (Akenine-Möller. and Haines 2019).

4.5 Ray tracing in video games

Ray tracing integration has recently become the new standard for triple A games. The advancements in Ray tracing techniques and the increasingly more powerful commercially available hardware have allowed for Ray tracing based effects to be present in a plethora of titles. Developers are eager to use this new technology, as it allows for more realistic visual outputs.

4.5.1 Uses of Ray Tracing

Ray tracing has been implemented in many different games, but not always for the same reason. This is because there are many different visual effects that Ray tracing enables, and different developers prioritized implementing different features for their games. The graph below (Figure 8) shows what features 11 different titles have chosen to implement (see Appendix 1 for data).

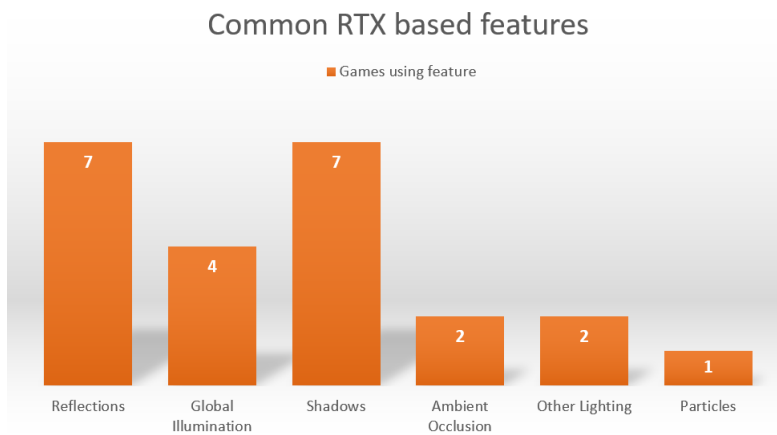


Figure 8: Ray tracing bound features across 11 triple A titles.

Reflections and Shadows were the two most popular Ray tracing features implemented; Global Illumination was also popular, however this was mainly in fully Ray tracing based engines.

There are two approaches at the moment when dealing with Ray tracing features in video game engines: either the engine is entirely based on Ray tracing or it uses a hybrid model between Rasterisation and Ray tracing.

The hybrid model is significantly more popular at the moment for triple A games (see Figure 9), as it diminishes the amount of computational power needed, making it more accessible to the public.

In addition, having the Ray tracing features as optional ensures that the potential audience is not restricted to only users with RTX graphics cards.

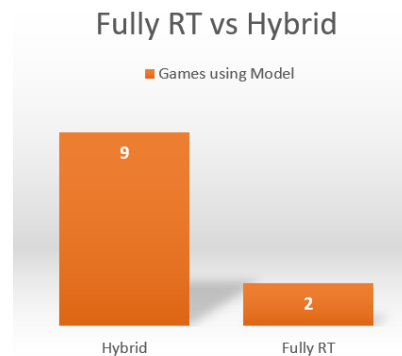


Figure 9: Engine models across 11 triple A titles.

4.5.2 Nvidia RTX: Real time Ray tracing

The highly desirable realistic visual outputs Ray tracing can create have been fueling research into methods of achieving them in real time. Progress has been steadily made over the years to reduce the amount of computational time required (e.g. Blender's Eevee engine is three times faster than its Cycles engine (Iatan 2019)).

The most substantial advancement came recently from Nvidia's introduction of the Turing architecture (see Figure 10) in graphics cards (Nvidia 2018). These architecture provides a number of RT Cores (number varies based on model), which are specifically designed to accelerate Ray tracing sufficiently for it to be viable in real time.



Figure 10: Turing TU102, TU104, TU106 Architecture (Nvidia 2018).

At the core of the hardware-accelerated ray tracing is an acceleration of the Bounding Volume Hierarchy traversal and Ray-triangle intersection testing; the latter is used to check if the light ray intersects with any object in the scene, while the former is used to define where this intersection happens.

Thanks to the commercial release of graphics cards using this architecture (named RTX cards), video game producers acquired freedom to implement Ray tracing in their games.

Bounding Volume Hierarchy

The Bounding Volume Hierarchy (BVH) acceleration structure is at the heart of what makes real time Ray tracing possible. It allows for ray-object intersections to be quickly identified, thanks to its tree-based structures (Nvidia 2018).

The BVH is constructed by increasingly smaller bounding boxes surrounding different geometry elements of a model (see Figure 11); these bounding boxes are grouped in a tree-based structure, with the bigger boxes at the higher levels. This allows for quick navigation of the different parts of the model, as large chunks can be rapidly discarded.

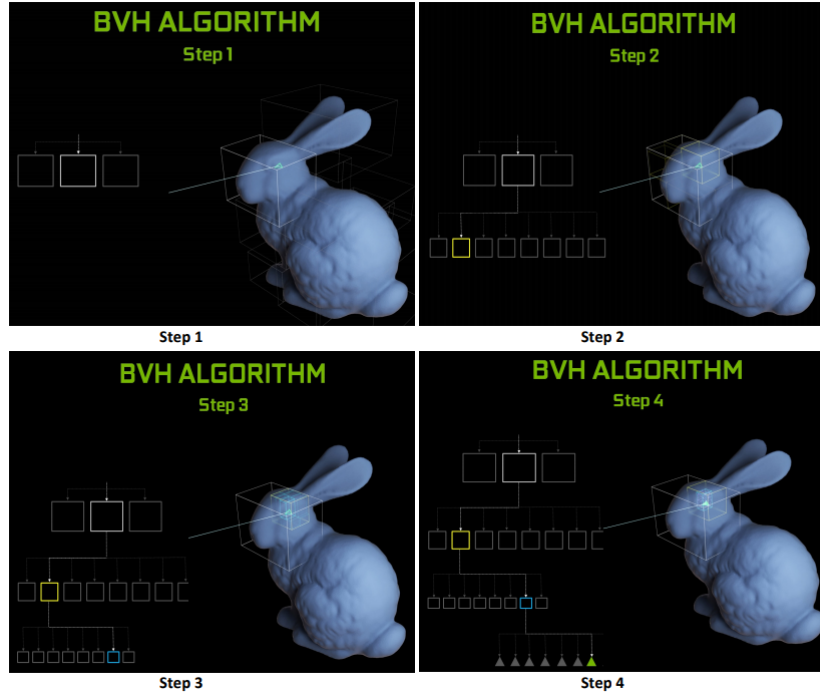


Figure 11: An example of navigating a BVH for a 3D rabbit model.

In the figure above, everything but the rabbit's head is discarded in step 1. Note that the amount of levels a BVH contains directly influences the degree of precision with which an intersection's position can be identified.

Reflections

Achieving real time reflections using Ray tracing remains a computationally expensive process, so the use of a Reflection Pipeline is recommended (see Figure 12).

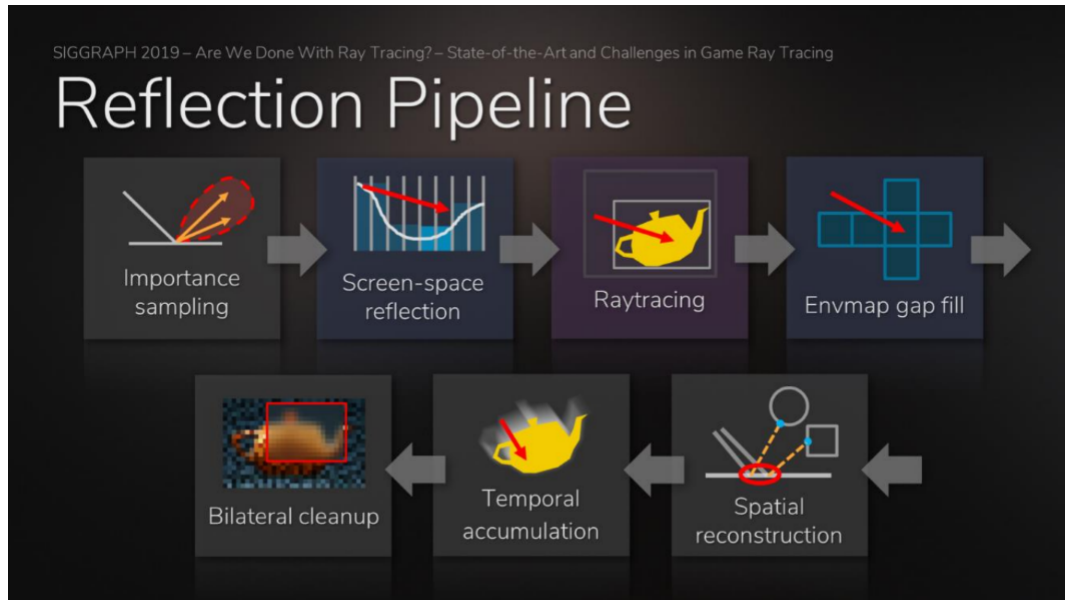


Figure 12: An example Reflection Pipeline (Barre-Brisebois 2019).

There are five different steps in most reflection pipelines (Barre-Brisebois 2019):

1. generate the rays. This is usually done using BRDF importance sampling, which ensures that more rays are allocated to more reflective materials;
2. perform screen-space intersections. This is a technique that focuses on reusing space data to calculate reflections(Unity Technologies 2017);
3. reconstruct the reflected image. This is done after all the intersections are found;
4. up-sample the image to full-resolution. Reflection Pipelines often use a down-scaled image as a starting point for faster calculations, but it has to be up-sampled again before completion;
5. clean up the image. This pipeline is likely to generate noise in the image, so cleanup using a form of a cross-bilateral filter is recommended.

In addition, some pipelines also add Temporal Anti-Aliasing (Korein and Badler 1983) at the end of the process, which further cleans up the output.

Shadows

Real time shadows are as a popular of a Ray tracing feature as real time reflections, but they are not as complicated nor computationally expensive.

Hard shadows are generated by launching rays from the ground towards the light source: if the ray collides with an object, then that spot is in shadow. To create soft shadows, the light source must be treated as an area light instead of a point (Barre-Brisebois 2019). Figure 13 presents a summary of the process with an example output.



Figure 13: How to generate real time shadows (Barre-Brisebois 2019).

Ambient occlusion

Ray tracing based ambient occlusion (RTAO) is meant to provide higher quality results when compared with non Ray tracing based techniques such as SSAO (Screen Space ambient occlusion). Comparatively to SSAO, RTAO leaves no "dark halo" around objects and carries visibility from geometry that is not visible (Liu 2018). The figure below shows the difference in quality between SSAO and RTAO.

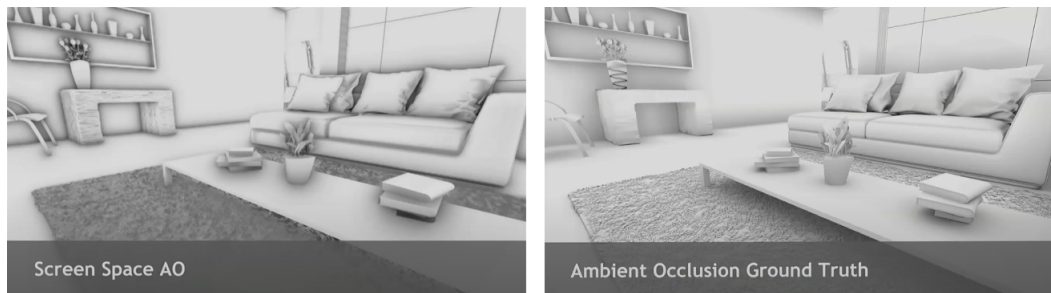


Figure 14: How to generate real time shadows (Barre-Brisebois 2019).

4.6 Ray tracing APIs

There are currently two ways of accessing the Ray tracing technology present in the Turing architecture (Nvidia 2018):

- using Microsoft’s DirectX Ray Tracing (DXR) extension, which fully integrates the ray tracing extension into the DirectX graphics API;
- using Vulkan, a royalty-free, cross-platform graphics API developed mainly by the Khronos Group.

OpenGL is also an option, but not a very appealing one. OpenGL does not directly support Ray tracing (Daniell 2019), but it allows the user to integrate Vulkan code.

Vulkan was developed later than DXR, with the first official release of its Ray tracing features dating to the last quarter of 2020, as a joint effort between the Khronos Group and Nvidia. It is similar to DXR, as the Khronos Group wanted it to be ”familiar to anyone who has used DirectX Ray tracing (DXR)” (Koch 2021), but it is designed to be *hardware-agnostic* (Subtil 2018).

While Vulkan might have functionality that DXR does not, its very recent release ensured that very few triple A titles have implemented and tested it yet. Its documentation is also still being constructed, which makes the development process of any program using it arduous. Lastly, many triple A developer companies currently use a DirectX engine, which is incompatible with the Vulkan Ray tracing features.

The extensive documentation provided by Nvidia and Microsoft for DXR and its popularity in the video game industry are the main reasons why it was chosen for this project.

4.7 Library design principles

Creating a library that has not been properly designed and documented is futile. A library that a user does not have an easy time using will only cause frustration and ensure that not many developers will attempt to implement it; because this project involves developing a library for implementing real time Ray tracing, which is a complex feature, its design and documentation must be of an excellent standard.

At the heart of the design of this project lay the S.O.L.I.D principles described by Robert C. Martin in his book ”Clean Architecture: A craftsman’s guide to software structure and design” (Martin 2017).

4.7.1 S.O.L.I.D Principles

The S.O.L.I.D principles are guidelines to follow for creating structures that tolerate change, are easy to understand and are the basis of components that can be used in many different software systems.

This makes them excellent for the design of this project’s software library, as they ensures that:

- the library can easily be updated if new technology is released;
- the user will have a comfortable experience with the library;
- the library is versatile enough to be used in many different game engines.

There are five different principles to follow, one for each letter (Janssen 2020, Martin 2017):

- Single Responsibility Principle, which means that a class should only have one responsibility and therefore should only ever change for a single reason. For example, a class that creates both the Acceleration structure and Ray tracing pipeline would not be following this principle, as it has two responsibilities; this class should be split in two classes, one that creates the Acceleration structure and one that creates the Ray tracing pipeline;
- Open-Closed Principle. This principle was first defined by Bertrand Meyer as "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." (Meyer 1997). This is very important for ensuring that the software is expandable, since it significantly reduces the workload for implementing new features;
- Liskov Substitution Principle. This principle was first introduced by Barbara Liskov in the conference "Data Abstraction" (Liskov 1987), and it focuses on the relationship between super-classes and sub-classes. It states that a super-class should be replaceable by one of its sub-classes. For example, if this project was to contain a super-class describing all types of shaders, instances of it should be replaceable by objects of sub-classes describing specific shaders (e.g. Ray generation shader);
- Interface Segregation Principle. This principle focuses on ensuring that interfaces should only contain methods which are relevant for all the classes linked to it. For example, if a interface for all the types of shaders was to include a *Generate Rays* method it would break this principle, as this method would not be useful for the Hit and Miss shader;
- Dependency Inversion Principle. This principle ensures that high level modules do not depend on low level ones, therefore they could be changed independently. This dependency is often eliminated by including another layer of abstraction between the two levels.

4.7.2 Documentation - Doxygen

For a library to be usable, it has to have a level of documentation. The standard for generating documentation for C++ code is to use Doxygen (Doxygen 2021). This software allows you to generate HTML and LaTeX manuals based on comments found in the source files (see Figure 15).

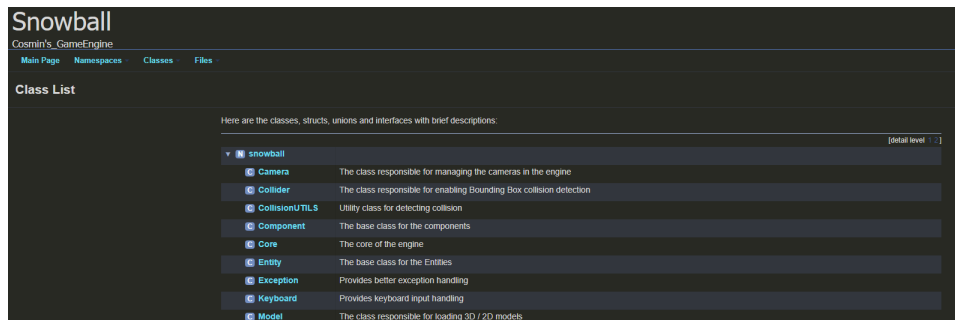


Figure 15: Example of part of the HTML documentation for a game engine.

This not only ensures that the code is properly commented, but it also provides users with an easy offline or online way to view the structure and in depth details of the library.

5 Design and Implementation

5.1 Defining the Library

This project is centred about automation of common tasks and facilitating development of Ray tracing bound features. The library is intended to be compatible with as many engines as possible. At the highest level, the library is designed to help with four different major tasks: initialising the DirectX 12 pipeline and set up the back end for real time Ray tracing, initializing the real time Ray tracing pipeline, creating and providing access to the acceleration structures needed (i.e. BVH management) and enabling path tracing.

The library will be developed using C++. It will use the DirectX 12 a DXR graphic APIs.

5.2 Compatibility with engines

For this project to be successful, the library has to be compatible with as many game engines as possible. This means that its requirements must be low enough to be accommodated by different architectures.

The library is based on DirectX 12 and the DXR API extension, which limits its compatibility to engines that already use DirectX 12 as their graphic API. This however does not restrict the library's viability by a significant amount, because more than a hundred games have already adapted DirectX 12 (PCGamingWiki 2021) since its official reveal in 2014 (Tyson 2014). This viability is further evidenced by the data in Appendix 1, which shows that most of the triple A games which have already adopted real time Ray tracing have done so using DXR (see Figure 16).

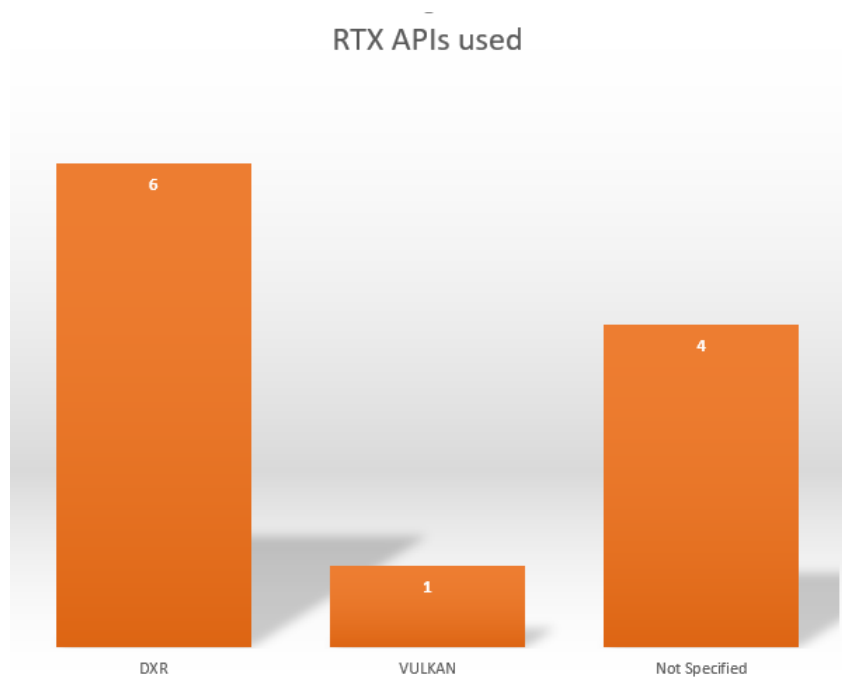


Figure 16: RTX APIs used across eleven triple A titles.

The library requires two fragments of information from the engine to produce an output:

- data of the models in the scene. This is used to create the acceleration structures and to enable path tracing;
- an output view port. The library will create an output and store it in a locally instantiated buffer (as per DirectX 12 rules), but it needs somewhere to copy it for it to be displayed.

Currently, there are two main types of engine architectures:

1. entity-component system based. This can be seen in engines such as Unity (Unity 2021), and it works by each game object having an instantiated renderer;
2. centralized system based. This is present in engines such as Unreal Engine (EpicGames 2021a), and it focuses on having a single central rendering module in the core of the engine.

For engines of type 1, the library should work. Although the renderer is bound to game objects, the data of the models and the view port are gathered in the engine's core; there they can easily be transferred over to the library.

The library should also be viable for engines of type 2. With the renderer being centralized, all of the model and view port data is also kept in the core (or at least referenced). This means that the data can again be easily transferred to the library. In addition, Unreal Engine has already implemented real time Ray tracing (EpicGames 2021b), which further evidences the library's compatibility.

5.3 DXR - DirectX Ray tracing

DXR was first released in October 2018 (van Rhyn 2018), making DirectX the first graphics API that allowed access to the Ray tracing capabilities of RTX graphic cards. Since then, many triple A developers have used DXR to implement Ray tracing features in their titles, such as Epic Games, EA and SEED (D3D Team 2018, see Appendix 1).

At the highest level, DXR adds four new features to DirectX 12 (D3D Team 2018):

1. an acceleration structure that represents the entire 3D virtual scene in a format optimal for traversal by the GPU. This is what ties in with the BVH acceleration provided by the Turing cores;
2. a new command list method *DispatchRays* that submits the work to the GPU;
3. a new set of shader types (i.e. ray-generation, closest-hit and miss shaders). This is what traces the path of light rays through the scene;
4. a Ray tracing pipeline state object, which provides details about the shaders and other Ray tracing workloads.

The early release and adoption of DXR ensures that the API is robust enough for comfortable use in new projects.

In addition, the documentation provided by Microsoft is excellent, with every method having an explanation and an example on the official website; there are also many examples and tutorials provided by Microsoft or other developers that have adopted the API already.

The API does however have some strict requirements (Smith 2014, Subtil 2018):

- it requires the engine to be already using DirectX 12, which is the latest version of the API and is not supported by older titles. It also implies that all the hardware and software requirements of DirectX 12 are applicable;
- it requires the Windows 10 SDK (version 10.0.19041.0), meaning it lacks the multi platform features of Vulkan. However it is worth noting that Windows currently owns a 32% market share, while Os X, Chrome OS and Other PC OS combined only own less than 10% (StatCounter 2021);
- all shaders must be written in High-Level Shading Language (HLSL), therefore shaders written in GL shading language (GLSL) need to be translated. This is not a particularly important issue for most engines, however some older ones might have some legacy code that needs to be adapted before being able to transition to the new API.

5.4 Library architecture

The library is designed to link directly with the core of the engine it is attached to. The overall class structures is showcased in the Figure 17 below.

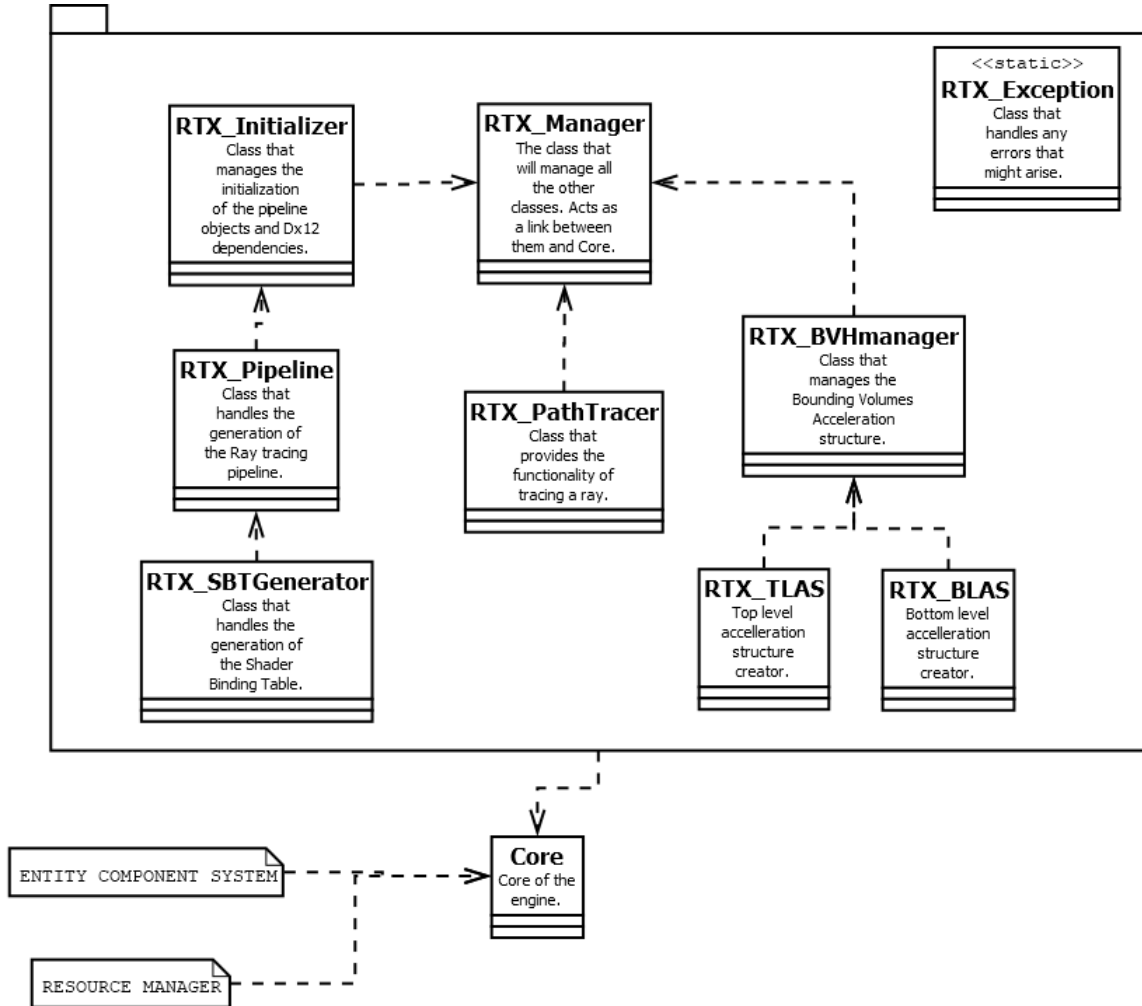


Figure 17: Top level class structure of the library.

To facilitate the communication between the engine and the library, there is one linking point between the library and the engine. The RTX_Manager class is this link (class overview in Appendix 3.1). It is responsible for acquiring and storing the data that the library requires; in addition, it also helps coordinate the different classes, as this is the only class the user will access directly. The RTX_Manager also acts as a link between the different classes, which is why all the others have a reference to the RTX_manager object instantiated. This link will allow the classes to share data between them. This class also incorporates the *Model* structure, which is used to store the geometry of the scene and its properties.

The only class that does not directly help with enabling real time Ray tracing is *RTX_Exception* (class overview in Appendix 3.8). Errors are bound to happen after prolonged use of the library. This class, accessible from everywhere within the library thanks to its *static* property, is responsible for catching these errors and helping the user fix them by providing an error message.

5.5 Initialising DirectX 12 and creating the back end

This process is kept almost the same among all implementations of DXR. This is because the API requires specific objects and pipelines to be initialized; however this process can often "run over 1000 lines of C++ code" (Akenine-Möller. and Haines 2019), therefore providing a standard implementation is guaranteed to lower the workload for developers.

To better understand this process, it can be split in eight different steps:

1. checking for an RTX compatible device. Real time Ray tracing is only possible on specific graphics cards, so this process ensures that the device meant to execute the code supports render passes, ray tracing and shader-resource view;
2. creating a *command list* and *command allocator*. The command list groups all the graphics commands the graphics card must execute (e.g. rendering), while the command allocator is used to allocate memory on the GPU;
3. creating a *root signature*. This object defines which resources are bound to the graphics pipeline, which means that it links the command list to the various shader resources it might depend upon;
4. creating the pipeline state. This involves compiling the main vertex and fragment shaders, which contain the high level GPU instructions, and creating the pipeline state object, which is used to read the properties of the pipeline;
5. creating the *swap chain*. DirectX 12 applications are not allowed to create an output directly on the render target, instead they must store it on a buffer first. A swap chain allows for the buffer to be copied on to the render target, as it is in charge of controlling the back buffer rotation;
6. creating the *descriptor heap*. Each object in DirectX 12 requires a descriptor to be initialized, which defines its properties. The descriptor heap provides memory allocation for these descriptors;
7. create a render target. As mentioned above, DirectX 12 applications require a buffer object to store their graphics output;
8. creating a *fence*. A fence is an object that allows for synchronization between the CPU and the GPU.

The class responsible for this process is the *RTX_Initializer* class (class overview in Appendix 3.2). In addition, it also acts as a coordinator for the *RTX_Pipeline* class, as they are both responsible for similar tasks. Having all the back end related creation tasks be accessible through the *Initializer* class adds an extra layer of abstraction for the user, which makes the library easier to navigate. The diagram in the Figure 18 shows how the user interaction with this process is simplified.

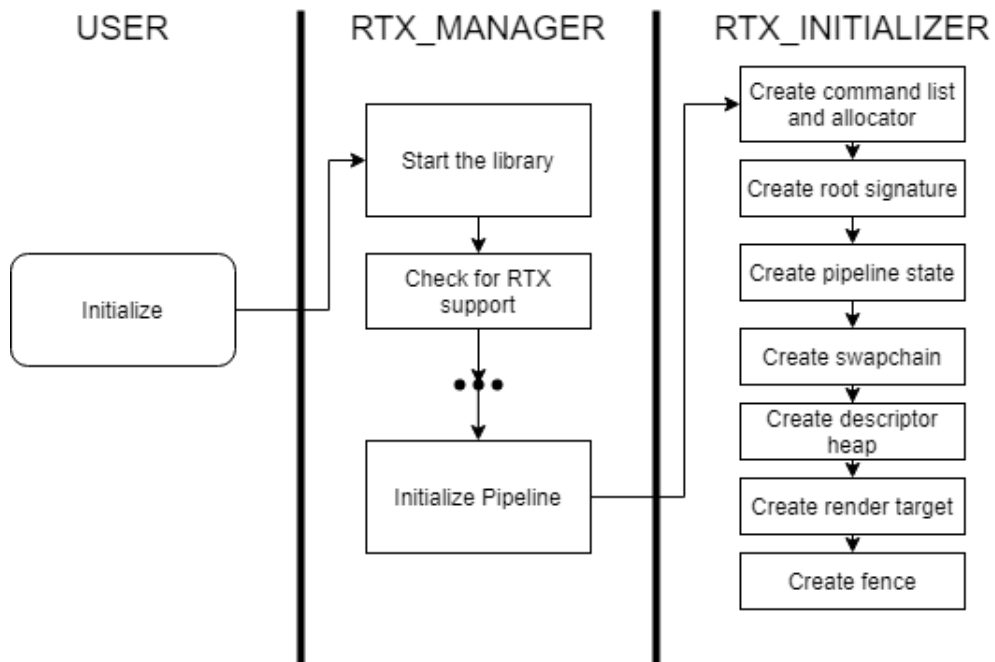


Figure 18: Diagram showcasing how the user interacts with the system for the initialization process. Notice that the user only ever accesses the RTX_Manager class.

The process for generating each of the Pipeline objects varies slightly between each one, but it mostly follows the following two step process:

- create a *descriptor*. This is a DirectX 12 object containing the properties of the Pipeline object to create;
- instruct the device to create the pipeline object based on the descriptor's data.

5.6 Creating the Ray tracing pipeline

The Ray tracing pipeline is mainly in charge of executing and managing shaders. A shader is a set of instructions (i.e. a program) to be executed on the graphics card (Gonzalez Vivo and Lowe 2015). Ray tracing requires multiple shaders to work, therefore DirectX created a default pipeline to support them (see Figure 19).

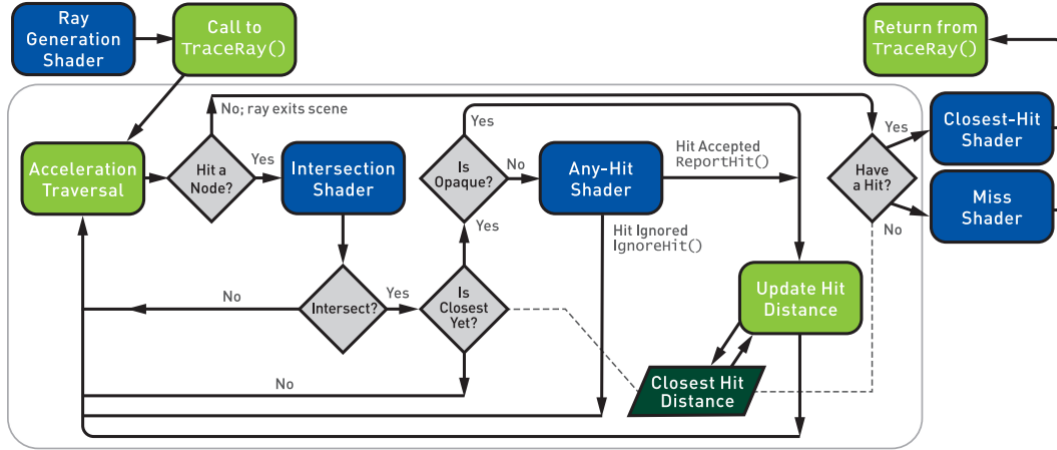


Figure 19: Simplified view of the new DirectX Ray tracing pipeline (Akenine-Möller and Haines 2019).

This pipeline consists of a series of control statements governing five different shaders:

- the ray generation shader, which creates the rays and starts the pipeline;
- the intersection shader, which defines how to handle all intersections. A default ray/triangle intersection shader is provided to enable the hardware acceleration aspect;
- the any-hit shader, which allows certain intersections (e.g. intersection with transparent object) to be ignored;
- the closest-hit shader, which dictates what should happen at the closest intersection along the ray;
- the miss shader, which defines how to handle when a shader does not hit any object.

Support for the creation of these new shaders has been added to HLSL, in the form of new data types, new textures and buffer resources, and new built-in functions.

The DirectX 12 pipeline handles the outputs created and provides the foundation for the application to work, while the Ray tracing pipeline focuses on enabling real time ray tracing. This pipeline is centred around the creation and uploading of the shaders needed (i.e. Ray generation shader, Miss shaders and Hit shaders).

The first step is to compile the shaders. This involves reading the source code of the shaders (written in High-Level Shading Language) and compiling them into a DirectX object. For this process, three objects are required:

- an *compiler library*. This acts as an interface and allows the shader source code to be stored locally;
- an *include handler*. Complex shaders might have dependencies on other files. The include handler ensures that these dependencies are respected;
- a *shader compiler*. This is what will compile the source code stored by the library into an executable ready to be sent to the graphics card.

Shaders will often require data to execute. For this data to be bound to the shader, a *root signature* is required for each one. These signatures are generated based on the previously compiled shader data and defined memory size / type. The signatures then need to be bound to the shader they relate to.

Lastly, all the data is grouped and sent to the graphics card. A state object is created, similarly to the generation of the DirectX 12 pipeline, which contains all the properties of the Ray tracing pipeline.

The RTX.Pipeline (class overview in Appendix 3.4) class is responsible for these DXR operations. As this process is relatively complex, it is grouped into the *createRaytracingPipeline()* method to simplify the user experience.

After the data is uploaded, a *shader binding table* (SBT) needs to be created to organize it. This is what associates the shaders with the geometry they need to operate on. The structure of a typical SBT can be seen below in Figure 20.

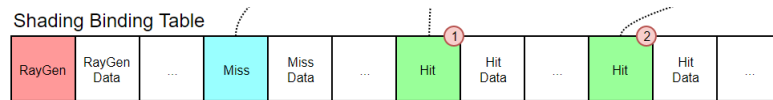


Figure 20: Diagram showcasing the structure of a typical SBT (Lefrançois 2021). Note how the data is contiguous to the shader it relates to.

The RTX_SBTGenerator (class overview in Appendix 3.3) is, as the name implies, the class responsible for creating the Shader Binding Table. To do this, the class makes use of the *SBTEntry* structure, which stores the entry point location and input data for a shader.

The entire process of the generating the Ray tracing Pipeline is summarized in the diagram below (Figure 21).

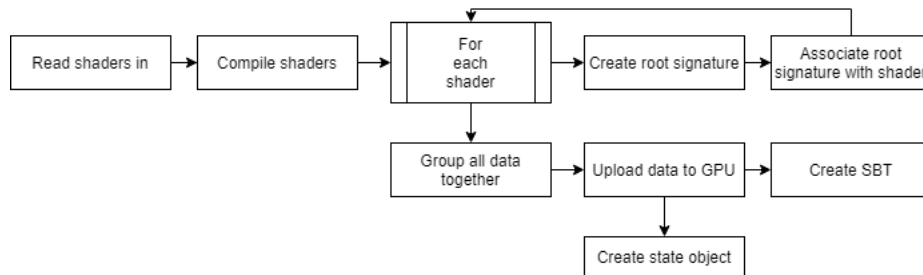


Figure 21: Diagram summarizing the creation of the DXR Ray tracing pipeline.

5.7 Creating the acceleration structures

As mentioned in the literature review section, hardware accelerated acceleration structures are a major part of what makes real time Ray tracing possible. The process for generating the acceleration structure in DXR is geometry independent, meaning that although the models its based on change from project to project, a standard implementation is still viable.

The acceleration structure in DXR is split in two parts: the bottom-level acceleration structure (BLAS) and the top-level acceleration structure (TLAS).

The BLAS contains the geometry that the rays will intersect with. Its generation is based on the *vertex buffers*, which are DirectX 12 objects created based on the description of the models. The creation starts with calculating the memory space needed for each BLAS; note that the amount of space needed changes from device to device, as they tend to include different overhead costs. Two buffers need to be allocated, a *scratch buffer* and a *result buffer*. The scratch buffer is used for temporary storage for when the BLAS is constructed; after, the data is the copied into the result buffer and the scratch buffer is deleted. Once the two buffers have been allocated, the GPU is instructed by the command list to generate the BLAS.

The TLAS can be thought of as a further optimization layer. The TLAS holds instances of bottom-level acceleration structures, so that the geometry can be reused in a scene without needing to regenerate the BLAS for it. The relation between top and bottom level acceleration structures can be seen in Figure 22. The process for generating a TLAS is very similar to the one for generating a BLAS, with the main difference being that it uses a BLAS as data instead of geometry.

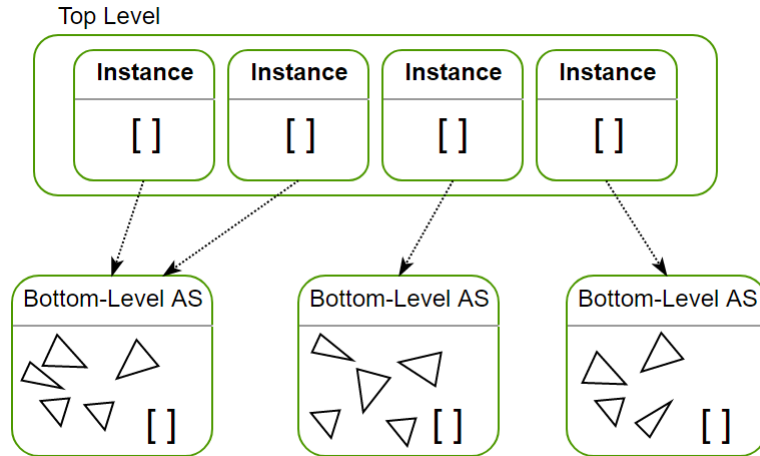


Figure 22: Diagram showcasing the relationship between the TLAS and the bottom-level acceleration structures it contains (Lefrançois 2021).

The process of creating the acceleration structure needed for real time Ray tracing is made more complex by having to generate a bottom-level and a top-level.

The `RTX_BVHManager` (class overview in Appendix 3.5) class acts as a coordinator for the `RTX_BLAS` (class overview in Appendix 3.7) and `RTX_TLAS` (class overview in Appendix 3.6) classes, which are each responsible for creating one of the levels of the acceleration structure. This not only allows for faster communication of data between the two classes, but further facilitates the use of the library as the user only has to call one method.

To generate a TLAS, the `RTX_TLAS` class makes use of the *Instance* structure (class overview in Appendix 3.6), which facilitates the storage of a bottom-level acceleration structure, together with its properties and transformations to apply.

5.8 Enabling path tracing

Ray tracing at its core is path tracing: the tracing of rays of light. This operation is very computationally expensive, yet can be significantly sped up by parallel processing, which is what makes it a perfect fit for being delegated to the GPU.

In DXR, path tracing is handled by the command list, which does a series of operations each frame (see Figure 23).

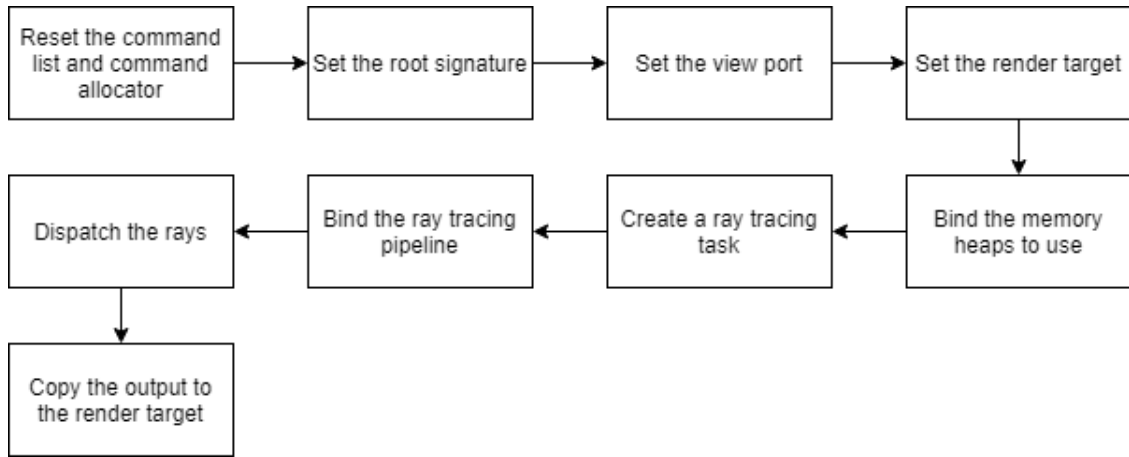


Figure 23: The different operations a command list must do each frame.

The dispatching of rays involves starting the Ray tracing task initialized, that specifies which Ray generation, Hit and Miss shaders to execute.

The class responsible for this process is `RTX_PathTracer` (class overview in Appendix 3.9). The class is relatively simple when compared to the others, since DXR does an excellent job at facilitating the process.

5.9 Using the library

The library needs to be accessed in the core class of an engine, where the rendering, frame update and initialization process is carried out. The diagram below shows the different calls that must be done to enable real time Ray tracing.

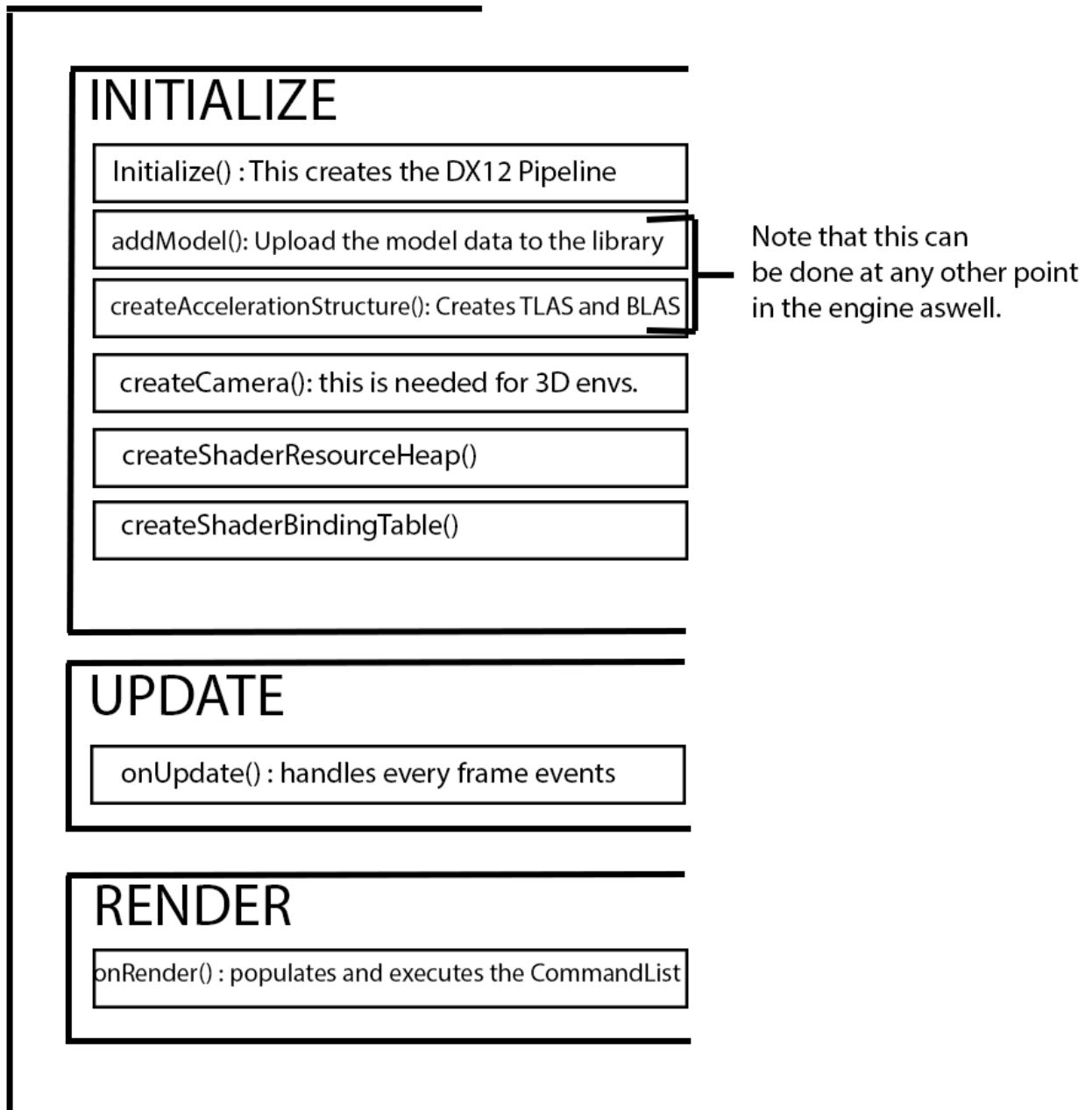


Figure 24: Calls needed for the library to produce a Ray traced output.

6 Testing

The most effective way to test this project is to implement the library into an existing engine. This would ensure that all the different components are executing as intended and, through the use of additional tools, the performance cost of implementing the library can be examined. In addition, this would also provide an example user experience, which can be used to check the difficulty of implementing the library.

6.1 Implementation in the engine

Using the library to generate a real time Ray tracing 3D output is a simple process for the user (see Figure 25).

```
rtxCORE = rtxCore->initialize(800, 800, hwnd, "shaders.hlsl", "Raygen.hlsl", "Miss.hlsl", "Hit.hlsl");
RTXSimplified::Vertex sample[] =
{
    {{0.0f, 0.25f * 1.f, 0.0f}, {1.0f, 1.0f, 0.0f, 1.0f}},
    {{0.25f, -0.25f * 1.f, 0.0f}, {0.0f, 1.0f, 1.0f, 1.0f}},
    {{-0.25f, -0.25f * 1.f, 0.0f}, {1.0f, 0.0f, 1.0f, 1.0f}}
};
rtxCORE->addModel(sample, 3);
rtxCORE->getBVHManager()->createAccelerationStructure();
rtxCORE->getInitializer()->getCommandList()->Close();
rtxCORE->getInitializer()->createRaytracingPipeline();
rtxCORE->getInitializer()->createPerInstanceConstantBuffers();
rtxCORE->getInitializer()->createCamera();
rtxCORE->getInitializer()->getPipeline()->createShaderResourceHeap();
rtxCORE->getInitializer()->getPipeline()->createShaderBindingTable();
```

Figure 25: The process of initializing the library.

As shown in the figure above, the initialization process consists of only eight commands:

1. initializing the DirectX 12 pipeline, which involves creating the various objects needed for a DirectX 12 application;
2. uploading the model data, which involves copying the data into a *Vertex* structure and storing that in the library;
3. creating the acceleration structures for the geometry uploaded. This groups up the creation of both the TLAS and BLAS;
4. creating the Ray tracing pipeline, which focuses on initializing all the DXR related objects;
5. creating the "per-instance" buffers, which are used to assign specific data to each BLAS instance;
6. creating a sample perspective camera, needed for any 3D scene;
7. creating the shader resource heap, which involves allocating the memory needed for each shader's data on the GPU;
8. creating the SBT, which is the structure needed to bind the shaders with their data.

After this initialization process has been completed, an output can be created by calling the *onRender()* command. This is typically done every frame and preceded by the *onUpdate()* command, which updates the properties of various objects (e.g. location of camera).

An example of an output created by the library can be seen below in Figure 26. This output's main focus is on demonstrating path tracing.

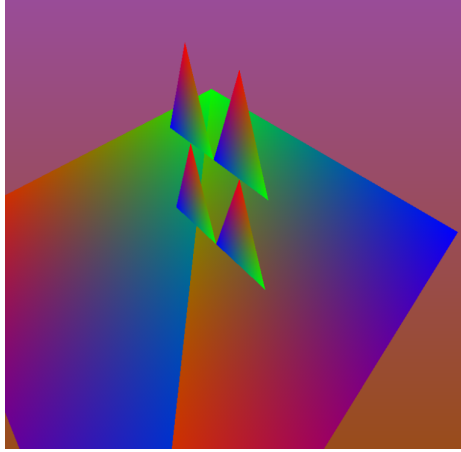


Figure 26: An output produced by using the library. It shows 4 triangles and a plane being seen through a perspective camera.

The output in the figure below demonstrates *per-instance data*, data associated with each instance of a BLAS (useful for model texturing), real time Ray traced shadows and *TLAS-refitting*, which updates the data of a TLAS (useful for animation).

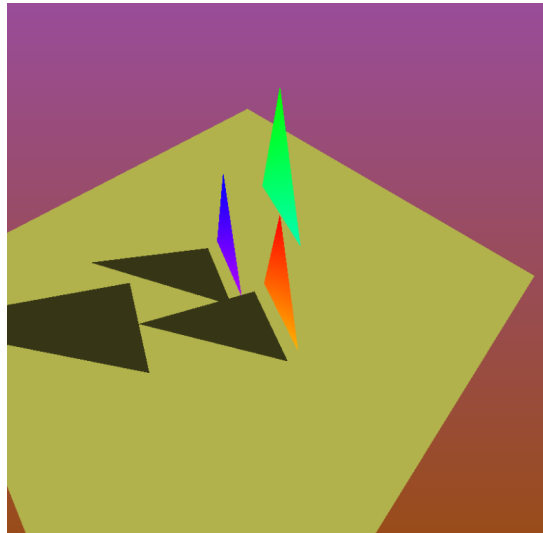


Figure 27: An output produced by using the library. It shows 3 triangles rotating and casting shadows on a plane. Each model is also coloured differently.

6.2 Performance costs

To analyze the performance costs of generating the output seen in Figure 32, the diagnostic tools provided by Microsoft Visual Studio Community 2019 Version 16.9.0 have been used. The table below shows the system used to execute the program.

Component type	Component Model
CPU	Intel(R) Core(TM) i9-9900KF CPU @ 3.60 GHz
GPU	NVIDIA GeForce RTX 2070 SUPER
RAM	16GB DDR4
OS	Windows 10 Home version 19042.928 64 bit

Table 1: Specifications of the system used to create the output shown in Figure 32.

Path tracing output - Figure 26

Figure 28 shows the CPU performance costs. The application had a very low initialization cost, which peaked at 6%. Following, the cost dropped to an insignificant amount. This is an expected result, as after the initial start up costs of creating some of the pipeline objects and the architecture of the library, the only CPU processes left are tied to synchronization between it and the GPU.

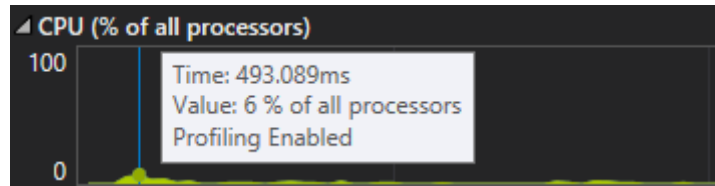


Figure 28: CPU usage for generating the output shown in Figure 26

Figure 29 shows the GPU performance costs. The GPU begins to operate after the initialization frame, as its tasked with the Ray tracing operation which repeats each frame. The task does not vary, so the cost is consistently kept around 7.5%, with fluctuations in the range of 0.1-0.3 %.

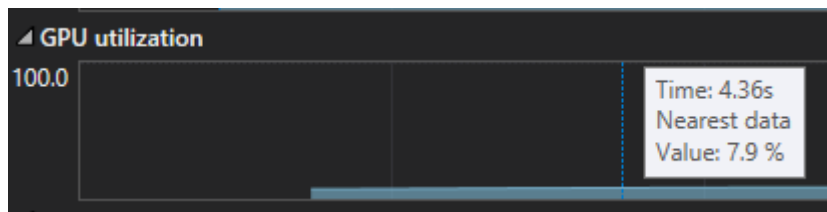


Figure 29: GPU usage for generating the output shown in Figure 26.

The frame time for the output was of 7ms, which means it succeeded in creating playable frame rates (30 frames per second). A more detailed report of the CPU/GPU usage can be found in Appendix 2.

Path tracing, Shadows, Animation and Per Instance data output - Figure 27

Adding the additional features had a very limited impact on the performance. The CPU costs were kept virtually the same, with an initial spike of 6% followed by a drop to an insignificant amount. The GPU costs increased by 0.5%, reaching an average of 8% with fluctuations still in the range of 0.1-0.3 %.

As the computational cost did not significantly increase, the frame time was still 7 ms, which meant that playable frame rates were still easily achievable (see Figure 30).

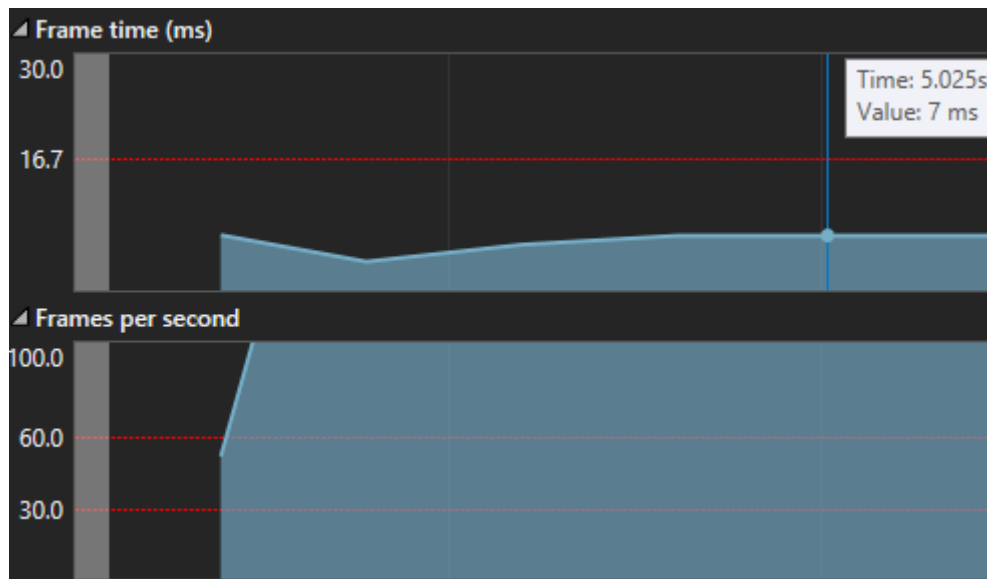


Figure 30: Frame time and Frames per second during the generation of the output shown in Figure 27.

7 Evaluation

7.1 Aims and objectives met

The aim of creating a library that allows developers to easily and rapidly create an environment for developing real time Ray tracing features has been met fully; likewise, the aim to test this library by implementing it into a game engine and producing a visual output has also been met.

This is evidenced by the completion of all the objectives set, which will be evaluated in the table below:

Objective	Evaluation
Conduct research into the background knowledge necessary for developing this project.	This objective has been completed. Research has been conducted to establish historical context for Ray tracing, what Ray tracing is and how does it work and how does it compare to rasterization. Research into techniques for implementing popular Ray tracing techniques have also been conducted. Lastly, research into library design principles has also been carried out.
Creating the library that allows developers to start working with real time Ray tracing.	This objective has been completed. DirectX 12 and DXR, the graphics API chosen for this project, have been researched and explained. The library has been designed according to the previously researched principles and implemented following said designs.
Implementing the library into an existing system to create a visual output.	This objective has been met. Evidence of this can be found in the testing section.
Evaluating the implementation of the library	This objective has been met. The user experience, viability and performance costs of the library have all been examined.

Table 2: Project’s objectives evaluation.

7.2 Future development

Although both the aims have been met, the project can easily be expanded on to improve the performance and the utility of the library. One of the design principles that the design of this library has adhered to is the Open-Closed Principle (detailed in the Literature Review section), which dictates that all modules should be open for extension. This means that all the different modules of the library can be expanded to include extra functionality.

The library should in future provide greater control to the user over the creation of the default pipeline components. At the moment the project creates standard implementation of many elements, which are designed to work in most cases; however, some specific needs may require modified parameters for both the DirectX 12 and the DXR pipeline. In addition, the library should allow the user to link an existing DirectX 12 pipeline with the DXR pipeline it creates; this would facilitate the integration of the library in engines that already use DirectX 12.

The library could also better support more advanced features. In its current state, the library allows developers to get a basic environment ready to start developing using real time Ray tracing and to generate real time Shadows, but it does not support them in creating complex features. The implementation of functions that help with the creation of complex features such as real time reflections, ambient occlusion or other popular features would further speed up the process for the developer.

This is also reflected in the visual output created using the library. At the moment, it shows a basic 3D environment with real time Shadows and different models; with more time allocated to this project it could better demonstrate some of the more advanced visual features made possible by real time Ray tracing.

Lastly, the library should be more extensively tested. At the moment, it has only been implemented into one engine, and said implementation has only been executed on one device. Since the aim of the project is to create a library that is compatible with as many game engines as possible, it should be implemented by multiple developers into their projects. This would not only reveal any potential errors and shortcomings that the library might have in specific situations, but it would also help better evaluate the user experience.

8 Conclusion

The project can be deemed a success. The aim of this project was to create a library that allowed users to more quickly and easily develop RTX based real time Ray tracing features. The project abstracts much of the complexity behind the creation of the pipelines necessary, and allows the user to easily create a basic ray traced output. The project also supports the user with implementing a couple of more complex features, such as real time shadows, animation and per instance data.

Nevertheless, there is always room for improvement. This project could benefit from providing more control over the initialization process to the user. Further support with complex features would also be helpful. Having multiple users experiment with this library would also help with testing the application.

9 Bibliography

Akenine-Möller., T. and Haines., E., 2019. Ray Tracing Gems. 1st ed. Apress

Appel, A., 1968. Some techniques for shading machine renderings of solids. Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring), 37-45.

Benthin, C., 2006. Realtime ray tracing on current CPU architectures. Doctoral Thesis. Universität Des Saarlandes.

Burke, B., 2019. Mood Lighting: How Metro Exodus Uses Ray Tracing to Heart-Pounding Effect — The Official NVIDIA Blog [online]. The Official NVIDIA Blog. Available from: <https://blogs.nvidia.com/blog/2019/02/15/metro-exodus-ray-tracing/> [Accessed 29 Apr 2021].

Burnes, A., 2020a. Cyberpunk 2077: Ray-Traced Effects Revealed, DLSS 2.0 Supported, Playable On GeForce NOW [online]. Nvidia. Available from: <https://www.nvidia.com/en-gb/geforce/news/cyberpunk-2077-ray-tracing-dlss-geforce-now-screenshots-trailer/> [Accessed 29 Apr 2021].

Burnes, A., 2020b. The Minecraft with RTX Beta Is Out Now! [online]. Nvidia. Available from: <https://www.nvidia.com/en-gb/geforce/news/minecraft-with-rtx-beta-out-now-download-play/#path-tracing> [Accessed 29 Apr 2021].

Burnes, A., 2020c. Watch Dogs: Legion Available Now with Ray Tracing and DLSS [online]. Nvidia. Available from: <https://www.nvidia.com/en-gb/geforce/news/watch-dogs-legion-pc-rtx-dlss-out-now/> [Accessed 29 Apr 2021].

Burnes, A., 2020d. Wolfenstein: Youngblood Update Adds Ray-Traced Reflections, NVIDIA DLSS and NVIDIA Highlights [online]. Nvidia. Available from: <https://www.nvidia.com/en-gb/geforce/news/wolfenstein-youngblood-ray-tracing-nvidia-dlss-highlights/> [Accessed 29 Apr 2021].

Burnes, A., 2019a. Shadow of the Tomb Raider Updates Adds Ray-Traced Shadows and DLSS [online]. Nvidia. Available from: <https://www.nvidia.com/en-gb/geforce/news/shadow-of-the-tomb-raider-nvidia-rtx-update-out-now/> [Accessed 29 Apr 2021].

Burnes, A., 2019b. Control: Multiple Stunning Ray-Traced Effects Raise The Bar For Game Graphics [online]. Nvidia. Available from: <https://www.nvidia.com/en-gb/geforce/news/control-rtx-ray-tracing-dlss-out-now/> [Accessed 29 Apr 2021].

Burnes, A., 2019c. Call of Duty: Modern Warfare Out Now with Ray Tracing, Ansel and Highlights [online]. Nvidia. Available from: <https://www.nvidia.com/en-gb/geforce/news/call-of-duty-modern-warfare-pc-out-now-with-ray-tracing/> [Accessed 29 Apr 2021].

CAULFIELD, B., 2018. What's the Difference Between Ray Tracing, Rasterization? — NVIDIA Blog [online]. The Official NVIDIA Blog. Available from: <https://blogs.nvidia.com/blog/2018/03/19/whatsdifference-between-ray-tracing-rasterization/> [Accessed 28 Apr 2021].

Barre-Brisebois, C., 2019. Are We Done With Ray Tracing? State-of-the-Art and Challenges in Game Ray Tracing.

D3D Team, 2018. Announcing Microsoft DirectX Raytracing! — DirectX Developer Blog [online]. DirectX Developer Blog. Available from: <https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/> [Accessed 28 Apr 2021].

Daniell, P., 2019. OpenGL / OpenGL ES Update.

Denham, T., 2021. Arnold, Corona & V-Ray Render Engines: What Exactly Do They Do? [online]. Concept Art Empire. Available from: <https://conceptartempire.com/arnold-corona-vray-render-engines/> [Accessed 10 May 2021].

Doxygen, 2021. Doxygen: Doxygen [online]. Doxygen.nl. Available from: <https://www.doxygen.nl/index.html> [Accessed 8 May 2021].

EpicGames, 2021a. Graphics Programming [online]. Docs.unrealengine.com. Available from: <https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Rendering/index.html> [Accessed 7 May 2021].

EpicGames, 2021b. Real-Time Ray Tracing [online]. Docs.unrealengine.com. Available from: <https://docs.unrealengine.com/en-US/RenderingAndGraphics/RayTracing/index.html> [Accessed 7 May 2021].

Ericson, C., 2005. Real-time collision detection. Amsterdam: Elsevier.

Fortnite Team, 2020. A New Light: Fortnite PC Now Supports Ray Tracing [online]. Epic Games' Fortnite. Available from: <https://www.epicgames.com/fortnite/en-US/news/a-new-light-fortnite-pc-now-supports-ray-tracing> [Accessed 29 Apr 2021].

Glassner, A., 1989. An introduction to ray tracing. 1st ed. San Francisco, Calif.: Morgan Kaufmann.

Gonzalez Vivo, P. and Lowe, J., 2015. The Book of Shaders [online]. The Book of Shaders. Available from: <https://thebookofshaders.com/01/> [Accessed 29 Apr 2021].

Gunther, J., Popov, S., Seidel, H. and Slusallek, P., 2007. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. 2007 IEEE Symposium on Interactive Ray Tracing.

Hapala, M., Davidovič, T., Wald, I., Havran, V. and Slusallek, P., 2013. Efficient stack-less BVH traversal for ray tracing. Proceedings of the 27th Spring Conference on Computer Graphics - SCCG '11.

Iatan, M., 2019. Cycles vs Eevee rendering – speed comparison [online]. RenderStreet blog. Available from: <https://blog.render.st/cycles-vs-eevee-rendering-speed-comparison/> [Accessed 28 Apr 2021].

Janssen, T., 2020. SOLID Design Principles [online]. Stackify. Available from: <https://stackify.com/solid-design-principles/> [Accessed 28 Apr 2021].

Koch, D., 2021. Vulkan Ray Tracing Final Specification Release [online]. The Khronos Group. Available from: <https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release> [Accessed 28 Apr 2021].

Korein, J. and Badler, N., 1983. Temporal anti-aliasing in computer generated animation. ACM SIGGRAPH Computer Graphics, 17 (3), 377-388.

Lee, T., 2019. The Best Rendering Software for CG Lighting for Animation - Academy of Animated Art [online]. Academy of Animated Art. Available from: <https://academyofanimatedart.com/the-best-rendering-software-for-cg-lighting-for-animation/> [Accessed 10 May 2021].

Lefrançois, M., 2021. DX12 Raytracing tutorial - Part 1 [online]. NVIDIA Developer. Available from: <https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-1> [Accessed 6 May 2021].

Liskov, B., 1987. PROGRAMMING WITH ABSTRACT DATA TYPES.

Liu, E., 2018. Ray Traced Ambient Occlusion.

Martin, R., 2017. Clean architecture. 1st ed. New Jersey: Pearson.

Yamazaki, M., 2021. In Depth: Which 3D Renderer is best? Compare features here! [online]. Toolfarm. Available from: https://www.toolfarm.com/tutorial/in_depth_3d_renderers/ [Accessed 8 May 2021].

Maxwell, 2021. Features — New, essential & advanced features of Maxwell render engine [online]. Maxwell Render. Available from: <https://maxwellrender.com/features/> [Accessed 10 May 2021].

Meyer, B., 1997. Object-oriented software construction. 1st ed. Upper Saddle River, N.J.: Prentice Hall PTR.

Nvidia, 2018. Get Started With Real-Time Ray Tracing [online]. NVIDIA Developer. Available from: <https://developer.nvidia.com/rtx/raytracing> [Accessed 28 Apr 2021].

NVIDIA, 2007. CUDA Zone [online]. NVIDIA Developer. Available from: <https://developer.nvidia.com/cuda-zone> [Accessed 8 May 2021].

NVIDIA, 2018. NVIDIA TURING GPU ARCHITECTURE [online]. NVIDIA. Available from: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf> [Accessed 28 Apr 2021].

Nvidia, 2021. Ray Tracing Essentials Part 2: Rasterization versus Ray Tracing [online]. video. Available from: <https://www.youtube.com/watch?v=ynCxnR1i0QY&t=68s> [Accessed 8 May 2021].

Park, S. and Baek, N., 2021. A Shader-Based Ray Tracing Engine. Applied Sciences, 11 (7), 3264.

PCGamingWiki, 2021. List of DirectX 12 games - PCGamingWiki PCGW - bugs, fixes, crashes,

mods, guides and improvements for every PC game [online]. Pcgamingwiki.com. Available from: https://www.pcgamingwiki.com/wiki/List_of_DirectX_12_games [Accessed 7 May 2021].

Razian, S. and MahvashMohammadi, H., 2017. Optimizing Raytracing Algorithm Using CUDA. Italian Journal of Science & Engineering, 1 (3), 167-178.

Redshift, 2021. Homepage [online]. Redshift3d.com. Available from: <https://www.redshift3d.com/> [Accessed 10 May 2021].

RenderNow, 2021. VFX And CGI Breakdown Of Avengers: Infinity War - RenderNow [online]. RenderNow. Available from: <https://rendernow.co.uk/vfx-and-cgi-breakdown-of-avengers-infinity-war/> [Accessed 10 May 2021].

Schied, C., 2021. Q2VKPT [online]. Brechpunkt.de. Available from: <http://brechpunkt.de/q2vkpt/#about> [Accessed 29 Apr 2021].

Smith, R., 2014. Microsoft Announces DirectX 12: Low Level Graphics Programming Comes To DirectX [online]. Anandtech.com. Available from: <https://www.anandtech.com/show/7889/microsoft-announces-directx-12-low-level-graphics-programming-comes-to-directx/2> [Accessed 28 Apr 2021].

StatCounter, 2021. Operating System Market Share Worldwide — StatCounter Global Stats [online]. StatCounter Global Stats. Available from: <https://gs.statcounter.com/os-market-share> [Accessed 28 Apr 2021].

SolidAngle, 2017. Arnold Renderer — Autodesk — 2017-scitech-award [online]. Arnoldrenderer.com. Available from: <https://www.arnoldrenderer.com/news/2017-scitech-award/> [Accessed 10 May 2021].

Subtil, N., 2018. Introduction to Real-Time Ray Tracing with Vulkan — NVIDIA Developer Blog [online]. NVIDIA Developer Blog. Available from: <https://developer.nvidia.com/blog/vulkan-raytracing/> [Accessed 28 Apr 2021].

Suffern, K., 2016. Ray tracing from the ground up. 1st ed. CRC Press.

Tyson, M., 2014. Microsoft introduces DirectX 12 at GDC 2014 [online]. HEXUS. Available from: <https://hexus.net/gaming/news/pc/67721-microsoft-introduces-directx-12-gdc-2014/> [Accessed 7 May 2021].

Unity, 2021. Entity Component System — Entities — 0.17.0-preview.41 [online]. Docs.unity3d.com. Available from: <https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/index.html> [Accessed 7 May 2021].

Unity Technologies, 2017. Unity - Manual: Screen Space Reflection [online]. Docs.unity3d.com. Available from: <https://docs.unity3d.com/560/Documentation/Manual/PostProcessing-ScreenSpaceReflection.html> [Accessed 28 Apr 2021].

van Rhyn, J., 2018. DirectX Raytracing and the Windows 10 October 2018 Update — DirectX Developer Blog [online]. DirectX Developer Blog. Available from: <https://devblogs.microsoft.com/directx/directx-raytracing-and-the-windows-10-october-2018-update/> [Accessed 28 Apr 2021].

Wald, I., Slusallek, P., Benthin, C. and Wagner, M., 2001. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20 (3), 153-165.

Wald, I., Purcell, T., Schmittler, J., Benthin, C. and Slusallek, P., 2003. Realtime Ray Tracing and its use for Interactive Global Illumination. *Eurographics 2003*.

Wald, I., Knoll, A., Johnson, G., Usher, W., Pascucci, V. and Papka, M., 2015. CPU Ray Tracing Large Particle Data with Balanced P-k-d Trees. *IEEE Visualization Conference*.

Whitted, T., 1980. An improved illumination model for shaded display. *Communications of the ACM*, 23 (6), 343-349.

Zlatuška, M. and Vlastimil, H., 2010. Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms. In: *WSCG. Plzen: Václav Skala - UNION Agency*, 69-76.

10 Appendix

10.1 Ray tracing features in games.

Game	Features	Engine Model	API	Source
Battlefield V	Reflections	Hybrid	DXR	Barre-Brisebois 2019
Quake 2	Global Illumination, Shadows, Reflections	RTX	Vulkan	Schied 2021
Metro Exodus	Global Illumination	Hybrid	N/A	Burke 2019
Cyberpunk 2077	Reflections, Shadows	Hybrid	DXR	Burnes 2020a
Shadow of the Tomb Raider	Shadows, Lighting	Hybrid	N/A	Burnes 2019a
Control	Reflections, Diffuse Lighting	Hybrid	N/A	Burnes 2019b
Minecraft	Global Illumination, Shadows, Reflections	RTX	DXR	Burnes 2020b
CoD: Modern Warfare	Shadows	Hybrid	DXR	Burnes 2019c
Watch Dogs Legion	Reflections	Hybrid	DXR	Burnes 2020c
Fortnite	Shadows, Reflections, Global Illumination	Hybrid	DXR	Fortnite Team 2020
Wolfenstein: Youngblood	Reflections	Hybrid	DXR	Burnes 2020d

Table 3: List of connections needed to display a textured 3D model

10.2 GPU and CPU usage report.

Child Level	Event Nam	CPU Start (CPU Durati	GPU Start (GPU Durati	Process Na	Thread ID	CPU Core
0	Present	1.39E+09	0	0	0	demo.exe	2828	
0		1.38E+09	0	1.39E+09	0	demo.exe	2828	
0		1.39E+09	0	1.39E+09	1.58E+08	demo.exe	2828	2
0		1.39E+09	0	1.39E+09	1.58E+08	demo.exe	2828	2
0	Present	1.39E+09	0	0	0	demo.exe	2828	2
0		1.39E+09	0	1.39E+09	0	demo.exe	2828	2
0		1.39E+09	0	1.39E+09	1.57E+08	demo.exe	2828	10
0		1.39E+09	0	1.39E+09	1.57E+08	demo.exe	2828	10
0		1.39E+09	1.57E+08	0	0	demo.exe	2828	2
0	Present	1.39E+09	0	0	0	demo.exe	2828	
0		1.39E+09	0	1.39E+09	0	demo.exe	2828	2
0		1.39E+09	0	1.39E+09	1.56E+08	demo.exe	2828	2
0		1.39E+09	0	1.39E+09	1.56E+08	demo.exe	2828	6
0		1.39E+09	1.56E+08	0	0	demo.exe	2828	4
0	Present	1.39E+09	0	0	0	demo.exe	2828	
0		1.39E+09	0	1.39E+09	0	demo.exe	2828	4
0		1.4E+09	0	1.4E+09	1.49E+08	demo.exe	2828	2
0		1.4E+09	0	1.4E+09	1.49E+08	demo.exe	2828	2
0		1.4E+09	1.49E+08	0	0	demo.exe	2828	2
0	Present	1.4E+09	0	0	0	demo.exe	2828	
0		1.4E+09	0	1.4E+09	0	demo.exe	2828	2
0		1.4E+09	0	1.4E+09	1.44E+08	demo.exe	2828	6
0		1.4E+09	0	1.4E+09	1.44E+08	demo.exe	2828	2
0	Present	1.4E+09	0	0	0	demo.exe	2828	2
0		1.4E+09	0	1.4E+09	0	demo.exe	2828	2
0		1.4E+09	0	1.4E+09	1.43E+08	demo.exe	2828	6
0		1.4E+09	0	1.4E+09	1.43E+08	demo.exe	2828	6
0		1.4E+09	1.43E+08	0	0	demo.exe	2828	2
0	Present	1.41E+09	0	0	0	demo.exe	2828	
0		1.4E+09	0	1.41E+09	0	demo.exe	2828	2
0		1.41E+09	0	1.41E+09	1.37E+08	demo.exe	2828	8
0		1.41E+09	0	1.41E+09	1.37E+08	demo.exe	2828	8
0	Present	1.41E+09	0	0	0	demo.exe	2828	8
0		1.41E+09	0	1.41E+09	0	demo.exe	2828	8
0		1.41E+09	0	1.41E+09	1.36E+08	demo.exe	2828	6
0		1.41E+09	0	1.41E+09	1.36E+08	demo.exe	2828	6
0		1.41E+09	1.35E+08	0	0	demo.exe	2828	2
0	Present	1.41E+09	0	0	0	demo.exe	2828	
0		1.41E+09	0	1.41E+09	0	demo.exe	2828	2
0		1.41E+09	0	1.41E+09	1.3E+08	demo.exe	2828	2
0		1.41E+09	0	1.41E+09	1.3E+08	demo.exe	2828	2
0	Present	1.41E+09	0	0	0	demo.exe	2828	2
0		1.41E+09	1.3E+08	0	0	demo.exe	2828	2
0		1.41E+09	0	1.41E+09	0	demo.exe	2828	2
0		1.42E+09	0	1.42E+09	1.29E+08	demo.exe	2828	6
0		1.42E+09	0	1.42E+09	1.28E+08	demo.exe	2828	8
0		1.42E+09	1.28E+08	0	0	demo.exe	2828	2
0	Present	1.42E+09	0	0	0	demo.exe	2828	

0	1.42E+09	0	1.42E+09	1.24E+08	demo.exe	2828	4
0	1.42E+09	0	1.42E+09	1.23E+08	demo.exe	2828	2
0 Present	1.42E+09	0	0	0	demo.exe	2828	1
0	1.42E+09	1.23E+08	0	0	demo.exe	2828	1
0	1.42E+09	0	1.42E+09	0	demo.exe	2828	1
0	1.42E+09	0	1.42E+09	1.22E+08	demo.exe	2828	8
0	1.42E+09	0	1.42E+09	1.22E+08	demo.exe	2828	8
0	1.42E+09	1.22E+08	0	0	demo.exe	2828	4
0 Present	1.43E+09	0	0	0	demo.exe	2828	
0	1.42E+09	0	1.43E+09	0	demo.exe	2828	4
0	1.43E+09	0	1.43E+09	1.17E+08	demo.exe	2828	8
0	1.43E+09	0	1.43E+09	1.17E+08	demo.exe	2828	8
0 Present	1.43E+09	0	0	0	demo.exe	2828	8
0	1.43E+09	0	1.43E+09	0	demo.exe	2828	8
0	1.43E+09	0	1.43E+09	1.16E+08	demo.exe	2828	4
0	1.43E+09	0	1.43E+09	1.16E+08	demo.exe	2828	4
0	1.43E+09	1.15E+08	0	0	demo.exe	2828	8
0 Present	1.43E+09	0	0	0	demo.exe	2828	
0	1.43E+09	0	1.43E+09	0	demo.exe	2828	8
0	1.43E+09	0	1.43E+09	1.15E+08	demo.exe	2828	6
0	1.43E+09	0	1.43E+09	1.15E+08	demo.exe	2828	6
0	1.43E+09	1.14E+08	0	0	demo.exe	2828	4
0 Present	1.43E+09	0	0	0	demo.exe	2828	
0	1.43E+09	0	1.43E+09	0	demo.exe	2828	4
0	1.43E+09	0	1.43E+09	1.1E+08	demo.exe	2828	4
0	1.44E+09	0	1.44E+09	1.09E+08	demo.exe	2828	2
0 Present	1.44E+09	0	0	0	demo.exe	2828	2
0	1.44E+09	0	1.44E+09	0	demo.exe	2828	2
0	1.44E+09	0	1.44E+09	1.09E+08	demo.exe	2828	2
0	1.44E+09	0	1.44E+09	1.08E+08	demo.exe	2828	4
0	1.44E+09	1.08E+08	0	0	demo.exe	2828	8
0 Present	1.44E+09	0	0	0	demo.exe	2828	
0	1.44E+09	0	1.44E+09	0	demo.exe	2828	8
0	1.44E+09	0	1.44E+09	1.03E+08	demo.exe	2828	10
0	1.44E+09	0	1.44E+09	1.03E+08	demo.exe	2828	10
0 Present	1.44E+09	0	0	0	demo.exe	2828	7
0	1.44E+09	0	1.44E+09	0	demo.exe	2828	7
0	1.44E+09	0	1.44E+09	1.01E+08	demo.exe	2828	1
0	1.44E+09	0	1.44E+09	1.01E+08	demo.exe	2828	1
0	1.44E+09	1.01E+08	0	0	demo.exe	2828	8
0 Present	1.45E+09	0	0	0	demo.exe	2828	
0	1.44E+09	0	1.45E+09	0	demo.exe	2828	8
0	1.45E+09	0	1.45E+09	95537975	demo.exe	2828	2
0	1.45E+09	0	1.45E+09	95350975	demo.exe	2828	4
0	1.45E+09	95053975	0	0	demo.exe	2828	0
0 Present	1.45E+09	0	0	0	demo.exe	2828	
0	1.45E+09	0	1.45E+09	0	demo.exe	2828	0
0	1.45E+09	0	1.45E+09	94175475	demo.exe	2828	2
0	1.45E+09	0	1.45E+09	94017475	demo.exe	2828	14
0	1.45E+09	93646275	0	0	demo.exe	2828	0

0 Present	1.46E+09	0	0	0 demo.exe	2828	
0	1.45E+09	0	1.46E+09	0 demo.exe	2828	0
0	1.46E+09	0	1.46E+09	88626975 demo.exe	2828	2
0	1.46E+09	0	1.46E+09	88471075 demo.exe	2828	2
0	1.46E+09	88100175	0	0 demo.exe	2828	2
0 Present	1.46E+09	0	0	0 demo.exe	2828	
0	1.46E+09	0	1.46E+09	0 demo.exe	2828	2
0	1.46E+09	0	1.46E+09	87276875 demo.exe	2828	4
0	1.46E+09	0	1.46E+09	87137475 demo.exe	2828	4
0	1.46E+09	86863475	0	0 demo.exe	2828	2
0 Present	1.46E+09	0	0	0 demo.exe	2828	
0	1.46E+09	0	1.46E+09	0 demo.exe	2828	2
0	1.46E+09	0	1.46E+09	81778475 demo.exe	2828	12
0	1.46E+09	0	1.46E+09	81580375 demo.exe	2828	12
0	1.46E+09	81305375	0	0 demo.exe	2828	8
0 Present	1.46E+09	0	0	0 demo.exe	2828	
0	1.46E+09	0	1.46E+09	0 demo.exe	2828	8
0	1.46E+09	0	1.46E+09	80097675 demo.exe	2828	15
0	1.46E+09	0	1.46E+09	79913275 demo.exe	2828	4
0	1.46E+09	79691575	0	0 demo.exe	2828	12
0 Present	1.47E+09	0	0	0 demo.exe	2828	
0	1.46E+09	0	1.47E+09	0 demo.exe	2828	12
0	1.47E+09	0	1.47E+09	74976875 demo.exe	2828	14
0	1.47E+09	0	1.47E+09	74815675 demo.exe	2828	14
0 Present	1.47E+09	0	0	0 demo.exe	2828	2
0	1.47E+09	0	1.47E+09	0 demo.exe	2828	2
0	1.47E+09	0	1.47E+09	74105675 demo.exe	2828	2
0	1.47E+09	0	1.47E+09	73943575 demo.exe	2828	4
0 Present	1.47E+09	0	0	0 demo.exe	2828	2
0	1.47E+09	0	1.47E+09	0 demo.exe	2828	2
0	1.47E+09	0	1.47E+09	73017475 demo.exe	2828	6
0	1.47E+09	0	1.47E+09	72868375 demo.exe	2828	6
0	1.47E+09	72673075	0	0 demo.exe	2828	12
0 Present	1.48E+09	0	0	0 demo.exe	2828	
0	1.47E+09	0	1.48E+09	0 demo.exe	2828	12
0	1.48E+09	0	1.48E+09	67504375 demo.exe	2828	4
0	1.48E+09	0	1.48E+09	67305275 demo.exe	2828	8
0	1.48E+09	67020475	0	0 demo.exe	2828	12
0 Present	1.48E+09	0	0	0 demo.exe	2828	
0	1.48E+09	0	1.48E+09	0 demo.exe	2828	12
0	1.48E+09	0	1.48E+09	64483975 demo.exe	2828	8
0	1.48E+09	0	1.48E+09	64296175 demo.exe	2828	8
0	1.48E+09	63814475	0	0 demo.exe	2828	13
0 Present	1.48E+09	0	0	0 demo.exe	2828	
0	1.48E+09	0	1.48E+09	0 demo.exe	2828	13
0	1.48E+09	0	1.48E+09	60661075 demo.exe	2828	12
0	1.48E+09	0	1.48E+09	60476975 demo.exe	2828	4
0 Present	1.48E+09	0	0	0 demo.exe	2828	2
0	1.48E+09	0	1.48E+09	0 demo.exe	2828	2
0	1.49E+09	0	1.49E+09	58482875 demo.exe	2828	12

0	1.49E+09	0	1.49E+09	58342775	demo.exe	2828	12
0	1.49E+09	58128675	0	0	demo.exe	2828	12
0 Present	1.49E+09	0	0	0	demo.exe	2828	
0	1.49E+09	0	1.49E+09	0	demo.exe	2828	12
0	1.49E+09	0	1.49E+09	53825975	demo.exe	2828	12
0	1.49E+09	0	1.49E+09	53573275	demo.exe	2828	2
0	1.49E+09	53339075	0	0	demo.exe	2828	2
0 Present	1.49E+09	0	0	0	demo.exe	2828	
0	1.49E+09	0	1.49E+09	0	demo.exe	2828	2
0	1.49E+09	0	1.49E+09	51786575	demo.exe	2828	15
0	1.49E+09	0	1.49E+09	51581475	demo.exe	2828	1
0	1.49E+09	51280175	0	0	demo.exe	2828	2
0 Present	1.5E+09	0	0	0	demo.exe	2828	
0	1.49E+09	0	1.5E+09	0	demo.exe	2828	2
0	1.5E+09	0	1.5E+09	46706275	demo.exe	2828	0
0	1.5E+09	0	1.5E+09	46398875	demo.exe	2828	13
0	1.5E+09	46093475	0	0	demo.exe	2828	2
0 Present	1.5E+09	0	0	0	demo.exe	2828	
0	1.5E+09	0	1.5E+09	0	demo.exe	2828	2
0	1.5E+09	0	1.5E+09	44819975	demo.exe	2828	0
0	1.5E+09	0	1.5E+09	44669075	demo.exe	2828	0
0	1.5E+09	44475875	0	0	demo.exe	2828	4
0 Present	1.5E+09	0	0	0	demo.exe	2828	
0	1.5E+09	0	1.5E+09	0	demo.exe	2828	4
0	1.5E+09	0	1.5E+09	40318875	demo.exe	2828	2
0	1.5E+09	0	1.5E+09	40179575	demo.exe	2828	2
0	1.5E+09	40033275	0	0	demo.exe	2828	2
0 Present	1.5E+09	0	0	0	demo.exe	2828	
0	1.5E+09	0	1.5E+09	0	demo.exe	2828	2
0	1.51E+09	0	1.51E+09	39423575	demo.exe	2828	5
0	1.51E+09	0	1.51E+09	39293975	demo.exe	2828	5
0	1.51E+09	39168475	0	0	demo.exe	2828	5
0 Present	1.51E+09	0	0	0	demo.exe	2828	
0	1.51E+09	0	1.51E+09	0	demo.exe	2828	5
0	1.51E+09	0	1.51E+09	38615475	demo.exe	2828	2
0	1.51E+09	0	1.51E+09	38511075	demo.exe	2828	2
0	1.51E+09	38394275	0	0	demo.exe	2828	2
0 Present	1.51E+09	0	0	0	demo.exe	2828	
0	1.51E+09	0	1.51E+09	0	demo.exe	2828	2
0	1.51E+09	0	1.51E+09	33538575	demo.exe	2828	2
0	1.51E+09	0	1.51E+09	33398075	demo.exe	2828	2
0	1.51E+09	33217575	0	0	demo.exe	2828	6
0 Present	1.51E+09	0	0	0	demo.exe	2828	
0	1.51E+09	0	1.51E+09	0	demo.exe	2828	6
0	1.51E+09	0	1.51E+09	32646775	demo.exe	2828	4
0	1.51E+09	0	1.51E+09	32496975	demo.exe	2828	3
0	1.51E+09	32264775	0	0	demo.exe	2828	2
0 Present	1.51E+09	0	0	0	demo.exe	2828	
0	1.51E+09	0	1.51E+09	0	demo.exe	2828	2
0	1.51E+09	0	1.51E+09	30213475	demo.exe	2828	2

10.3 Class diagrams

10.3.1 RTX_Manager

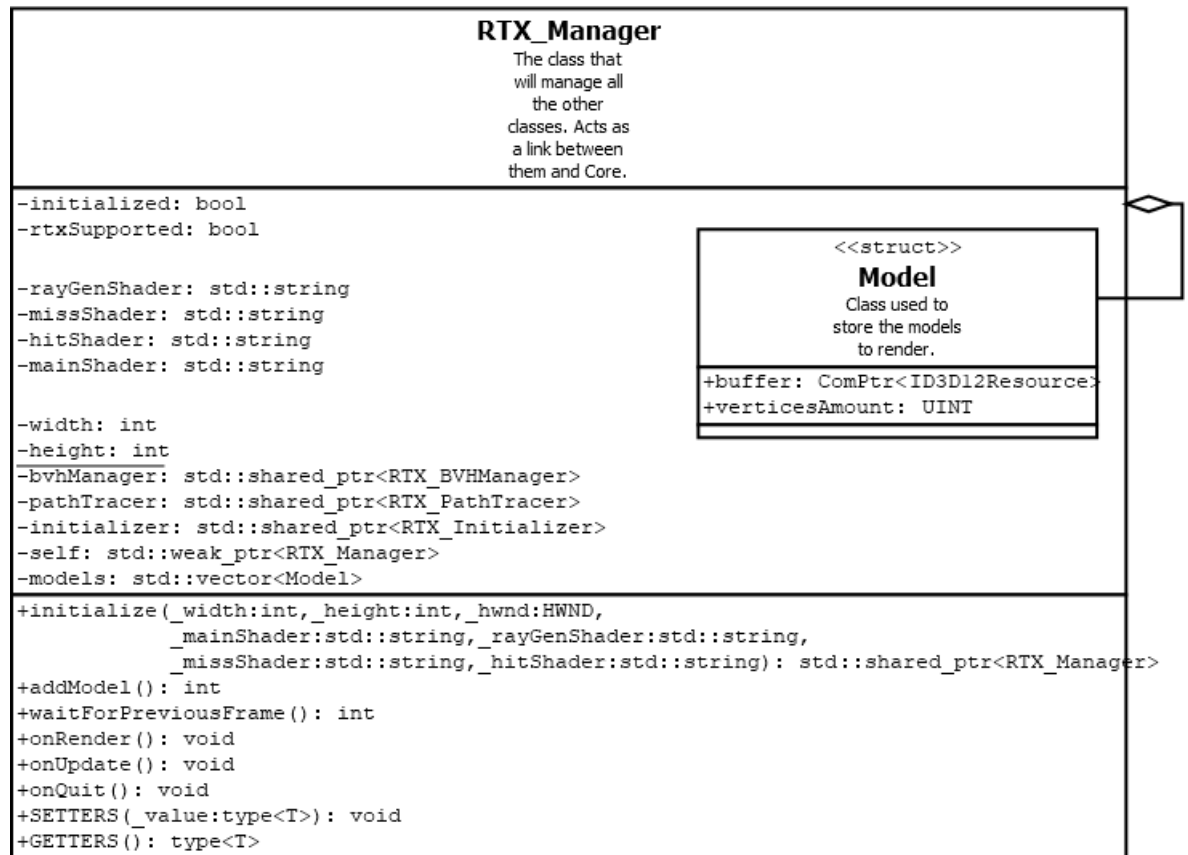


Figure 31: Overview of the RTX_Manager class.

10.3.2 RTX_Initializer

RTX_Initializer Class that manages the initialization of the pipeline objects and Dx12 dependencies.
<pre> - pipeline: std::shared_ptr<RTX_Pipeline> - rtxSupported: bool - rtxDevice: ComPtr<ID3D12Device5> - commandQueue: ComPtr<ID3D12CommandQueue> - factory: ComPtr<IDXGIFactory4> - swapChain: ComPtr<IDXGISwapChain3> - viewport_width: int - viewport_height: int - rtxManager: std::shared_ptr<RTX_Manager> - descriptorHeapSize: UINT - renderTarget: ComPtr<ID3D12Resource> - commandAllocator: ComPtr<ID3D12CommandAllocator> - rtStateObject: ComPtr<ID3D12StateObject> - rtStateObjectProps: ComPtr<ID3D12StateObjectProperties> - outputResource: ComPtr<ID3D12Resource> - rootSignature: ComPtr<ID3D12RootSignature> - viewport: CD3DX12_VIEWPORT - scissorRect: CD3DX12_RECT - rtvHeap: ComPtr<ID3D12DescriptorHeap> - pipelineState: ComPtr<ID3D12PipelineState> - commandList: ComPtr<ID3D12GraphicsCommandList5> - fenceEvent: HANDLE - fence: ComPtr<ID3D12Fence> - fenceValue: UINT64 - frameIndex: UINT - cameraBuffer: ComPtr<ID3D12Resource> - constantHeap: ComPtr<ID3D12DescriptorHeap> - cameraBufferSize: uint32_t - globalConstantBuffer: ComPtr<ID3D12Resource> - perInstanceConstantBuffers: std::vector<ComPtr<ID3D12Resource>> - createDevice(): int - createAdapter(_factory:IDXGIFactory2*, _adapter:IDXGIAdapter1**): int - createCommandQueue(): int - createSwapChain(): int - createDescriptorHeaps(): int - createFrameResources(): int - createCommandAllocator(): int - createRTOutput(): int - createPipelineState(): int - createFence(): int +createCamera(): int +checkRTXSupport(): int +createPipeline(): int +createRaytracingPipeline(): int +prepareAssetLoading(): int +int(): updateCameraBuffer +createGlobalConstantBuffer(): int +createPerInstanceConstantBuffers(): int </pre>

Figure 32: Overview of the RTX_Initializer class.

10.3.3 RTX_SBTGenerator

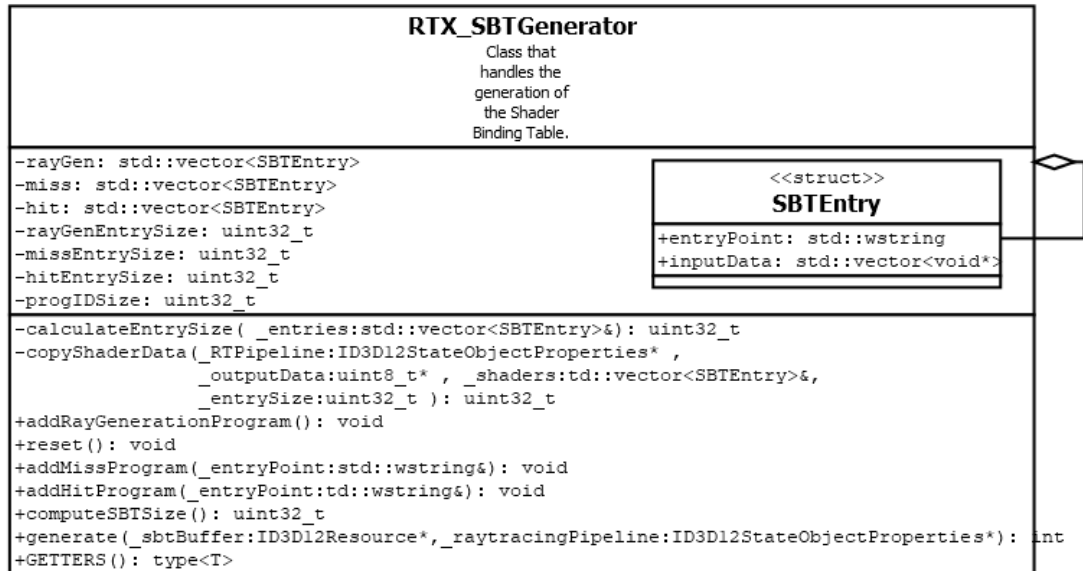


Figure 33: Overview of the RTX_SBTGenerator class.

10.3.4 RTX_Pipeline

RTX_Pipeline
Class that handles the generation of the Ray tracing pipeline.
<pre> -maxAttributeSizeInBytes: UINT -maxRecursionDepth: UINT -maxPayloadSizeInBytes: UINT -libraries: std::vector<Library> -hitgroups: std::vector<HitGroup> -rootSigAssociations: std::vector<RootSignatureAssociation> -compiler: IDxcCompiler* -library: IDxcLibrary* -includeHandler: IDxcIncludeHandler* -rayGenLib: ComPtr<IDxcBlob> -hitLib: ComPtr<IDxcBlob> -missLib: ComPtr<IDxcBlob> -defaultGlobalSignature: ID3D12RootSignature* -defaultLocalSignature: ID3D12RootSignature* -rtxManager: std::shared_ptr<RTX_Manager> -rayGenSig: ComPtr<ID3D12RootSignature> -hitSig: ComPtr<ID3D12RootSignature> -missSig: ComPtr<ID3D12RootSignature> -srvUavHeap: ComPtr<ID3D12DescriptorHeap> -SBTGenerator: RTX_SBTGenerator -sbtStorage: ComPtr<ID3D12Resource> -compileShaderLib(_shaderFile:std::string): IDxcBlob* -addLibrary(_library:IDxcBlob*,_symbols:std::vector<std::wstring>&): void -createRayGenSig(): ComPtr<ID3D12RootSignature> -createMissSig(): ComPtr<ID3D12RootSignature> -createHitSig(): ComPtr<ID3D12RootSignature> -buildShaderExportList(): int -addRootSignatureAssociation(_rootSig:ID3D12RootSignature* , _symbols:std::vector<std::wstring>&): int -createDescriptorHeap(_count:uint32_t,D3D12_DESCRIPTOR_HEAP_TYPE:_type): ID3D12DescriptorHeap +createBuffer(_device:ID3D12Device*,_size:uint64_t, _flags:D3D12_RESOURCE_FLAGS,): ID3D12Resource* +addHitGroup(_hitGroupName:std::wstring&, _closestHitSymb:std::wstring&, _anyHitSymb:std::wstring&,_intersectionSymb:std::wstring&): int +addRootSignatureAssociation(): int +createDefaultRootSignature(): int +createShaderLibraries(): int +addLibraries(): int +createShaderSignatures(): int +generate(): ID3D12StateObject* +createShaderResourceHeap(): int +createShaderBindingTable(): int +GETTERS(): type<T> +SETTERS(_value:type<T>): void </pre>

Figure 34: Overview of the RTX_Pipeline class.

10.3.5 RTX_BVHManager

RTX_BVHmanager Class that manages the Bounding Volumes Acceleration structure.
<pre>+defaultHeapProperties: static const D3D12_HEAP_PROPERTIES +uploadHeapProperties: static const D3D12_HEAP_PROPERTIES -BLASmanager: RTX_BLAS -TLASmanager: RTX_TLAS -buffer: AccelerationStructureBuffers -rtxManager: std::shared_ptr<RTX_Manager> -instances: std::vector<std::pair<ComPtr<ID3D12Resource>, DirectX::XMMATRIX> -TLASBuffers: AccelerationStructureBuffers -bottomLevelAS: ComPtr<ID3D12Resource> -createBLAS(_vertexBuffers: (std::vector<std::pair<ComPtr<ID3D12Resource>, uint32_t>>): AccelerationStructureBuffers -createTLAS(_instances: std::vector<std::pair<ComPtr<ID3D12Resource>, DirectX::XMMATRIX>>&): int +createBuffer(_device: ID3D12Device*, _size: uint64_t, _flags: D3D12_RESOURCE_FLAGS, _initState: D3D12_RESOURCE_STATES, _heapProps: D3D12_HEAP_PROPERTIES): ID3D12Resource* +createAccelerationStructure(): int +GETTERS(): type<T> +SETTERS()(_value: type<T>): void</pre>

Figure 35: Overview of the RTX_BVHManager class.

10.3.6 RTX_TLAS

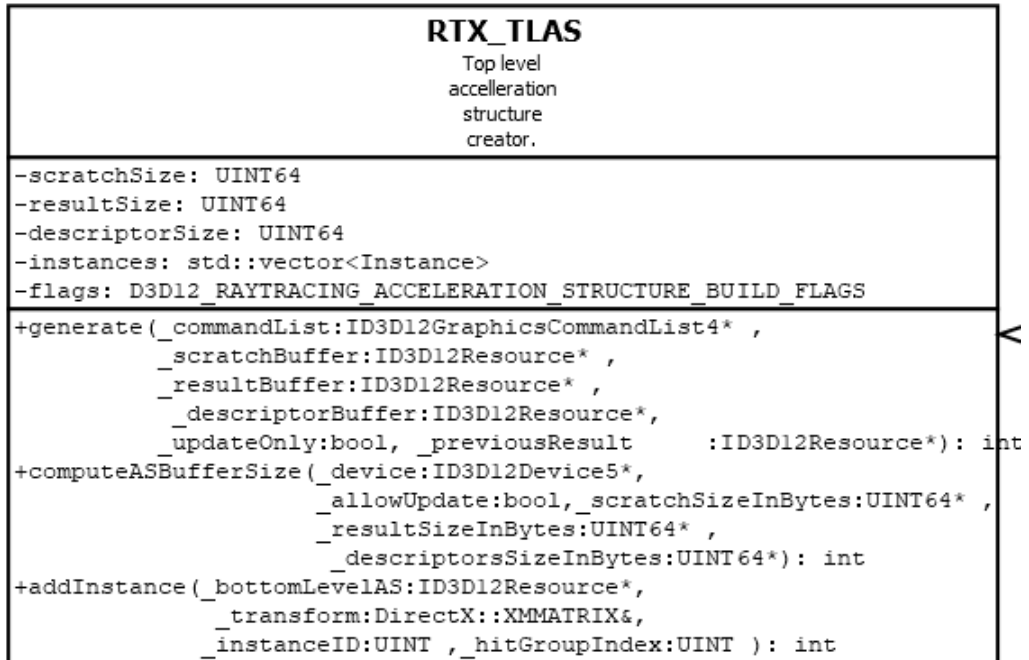


Figure 36: Overview of the RTX_TLAS class.

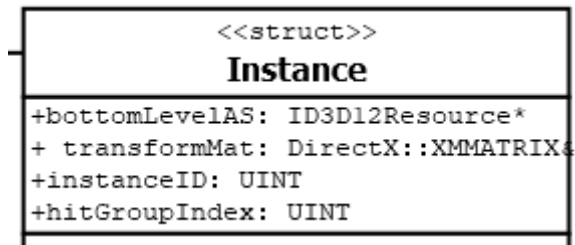


Figure 37: Overview of the Instance structure.

10.3.7 RTX_BLAS

RTX_BLAS Bottom level acceleration structure creator.
<pre>-scratchSize: UINT64 -resultSize: UINT64 -vertexBuffers: std::vector<D3D12_RAYTRACING_GEOMETRY_DESC> -flags: D3D12_RAYTRACING_ACCELERATION_STRUCTURE_BUILD_FLAGS +addVertexBufferLimited(_vertexBuffer:ID3D12Resource*, _vertexOffsetInBytes:UINT64 , _vertexCount:uint32_t , _vertexSizeInBytes:UINT, _indexBuffer:ID3D12Resource*, _indexOffsetInBytes:UINT64 , _indexCount:uint32_t , _transformBuffer:ID3D12Resource* , _transformOffsetInBytes:UINT64, _isOpaque :bool): int +generate(_commandList:ID3D12GraphicsCommandList4*, _scratchBuffer:ID3D12Resource* , _resultBuffer:ID3D12Resource* , _updateOnly:bool , _previousResult:ID3D12Resource*): int +computeASBufferSize(_device:ID3D12Device5* , allowUpdate:bool _,_scratchSizeInBytes:UINT64* _resultSizeInBytes:UINT64*): int</pre>

Figure 38: Overview of the RTX_BLAS class.

10.3.8 RTX_Exception

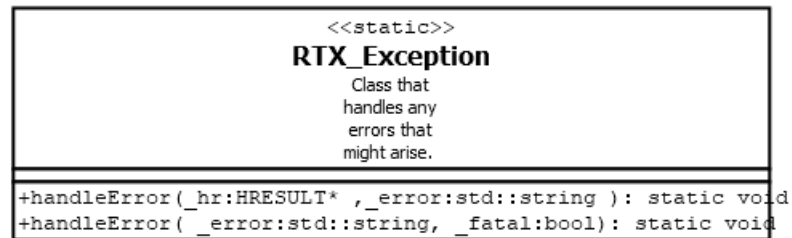


Figure 39: Overview of the RTX_Exception class.

10.3.9 RTX_PathTracer

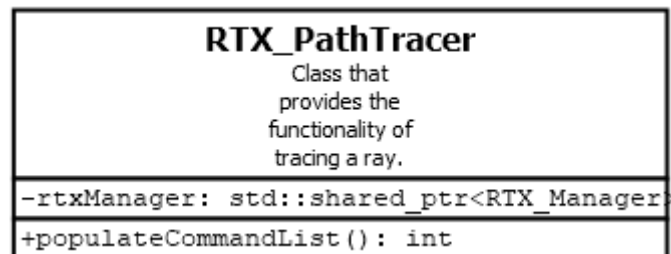


Figure 40: Overview of the RTX_PathTracer class.